Global Consensus through Local Synchronization

Sung-Shik T.Q. Jongmans and Farhad Arbab

Centrum Wiskunde and Informatica, Amsterdam, Netherlands
{jongmans,farhad}@cwi.nl

Abstract. Coordination languages have emerged for the specification and implementation of interaction protocols among concurrent entities. Currently, we are developing a code generator for one such a language, based on the formalism of constraint automata (CA). As part of the compilation process, our tool computes the CA-specific synchronous product of a number of CA, each of which models a constituent of the protocol to generate code for. This ensures that implementations of those CA at run-time reach a consensus about their global behavior in every step. However, using the existing product operator on CA can be practically problematic. In this paper, we provide a solution by defining a new, local product operator on CA that avoids those problems. We then identify a sufficiently large class of CA 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 entities (services, threads, etc.). This class of languages includes Reo [1,2], a graphical dataflow language for compositional construction of *connectors*: communication media through which entities can interact with each other. Figure 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 common nodes), users can construct arbitrarily complex connectors.

To implement and use connectors in real applications, one must derive implementations from their graphical specification [3,4,5,6,7,8,9], as precompiled executable code or using a run-time interpretation engine. Roughly two implementation approaches currently exist. In the *distributed approach*, 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*, one computes the behavior of a connector as a whole, implements this behavior, and runs this implementation sequentially as a centralized system.

Currently, we are developing a Reo-to-Java code generator using the centralized approach based on the formalism of *constraint automata* (CA) [10]. On input of a graphical connector specification (as an XML file), our tool automatically

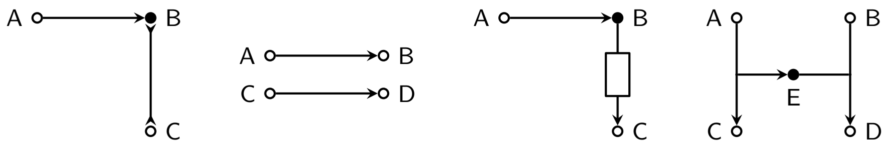

Fig. 1. Four example connectors. Open circles represent *boundary nodes*, on which entities perform I/O-operations; filled circles represent nodes for internal routing. Every connector in this figure consists of two *primitives* (i.e., minimal subconnectors); the pairs of primitives in the first, third, and fourth connector have one common node.

generates code in four steps. *First*, it extracts from the specification a list of the channels constituting the specified connector. *Second*, it consults a database to find for every channel in the list a "small" CA that formally describes the behavior of that particular channel. *Third*, it computes the product of the CA in the constructed collection to obtain one "big" CA describing the behavior of the whole connector. *Fourth*, it feeds a data structure representing that big CA to a template. Essentially, this template is an incomplete Java class with "holes" that need be "filled" (with information from the data structure). The class generated in this way implements Java's `Runnable` interface. This means that a Java virtual machine can execute the implemented `run` method (declared in `Runnable` and generated by our tool), which simulates the big CA computed in the third step, sequentially in a separate thread (details appear elsewhere [4]).

Problem. Computing one big CA (the third step of the centralized approach) and afterward translating it to sequential code (the fourth step) can be problematic: at run-time, the generated implementation may unnecessarily restrict parallelism among independent transitions.[1] The problem is implementing such a big CA using exactly one thread: single-threaded programs cannot execute multiple independent transitions simultaneously, but instead, they force those transitions to execute one after the other (see Section 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 this problem is to *not* compute one big CA but generate code directly for each of the small CA instead, essentially moving from the centralized approach to the distributed approach: the implementations of the small CA compute the product operators between them at run-time instead of at compile-time. Although this approach solves the stated problem—independent transitions can execute simultaneously—the necessary distributed algorithms for run-time product computation may inflict a substantial amount of overhead.

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

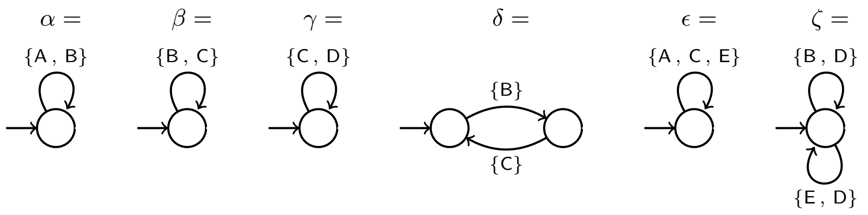

Fig. 2. Port automata, denoted by α, β, γ, δ, ϵ, and ζ, describing the behavior of the primitives constituting the example connectors in Figure 1: α and β model the primitives in the first connector, α and γ the primitives in the second, α and δ the primitives in the third, and ϵ and ζ the primitives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

Contribution. This paper provides a better solution to the stated problem by offering a middle ground between centralized and distributed approaches, wherein some subsets of the constituent automata are statically composed to comprise a distributed system of locally centralized automata. Typically, each locally centralized automaton interacts/synchronizes with few other such automata for its transitions, while it represents the composition of a subset of the constitutent 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 computation at run-time requires only relatively simple distributed algorithms—CA 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 [8].

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

2 Preliminaries: Port Automata

Many formalisms exist for mathematically defining the semantics of connectors [11]; our code generator, for instance, relies on constraint automata (CA). In this paper, however, we adopt a simplification of CA, called *port automata* (PA) [12]. We prefer PA, because they allow us to focus on the core of our problem (synchronization of communication) without getting distracted by those details of CA (the data exchanged in communication) irrelevant to our present purpose. The results in this paper straightforwardly carry over from PA to CA.

A PA consists of a finite set of states and transitions between them, each of which has a set of *ports* as label. A transition represents an execution step of a connector, from one internal configuration to the next, where synchronous interaction occurs on the ports labeling that transition. Let $\mathbb{P}\textsc{ort}$ and $\mathbb{S}\textsc{tate}$ denote global sets of ports and states (see [13, Appendix A] for formal definitions).

Definition 1 (Universe of port automata). *The universe of* PA*, denoted by* $\mathbb{P}\textsc{a}$ *and typically ranged over by* α*,* β*, or* γ*, is the largest set of tuples* $(Q\,,\mathcal{P}\,,\longrightarrow\,,\imath)$ *where:*[2]

- $Q \subseteq \mathbb{S}\textsc{tate}$*;* *(states)*
- $\mathcal{P} \subseteq \mathbb{P}\textsc{ort}$*;* *(ports)*
- $\longrightarrow \subseteq Q \times \wp(\mathcal{P}) \times Q$*;* *(transitions)*
- *and* $\imath \in Q$*.* *(initial state)*

Figure 2 shows example PA. For instance, the $\{\mathsf{A}\,,\mathsf{B}\}$-transition of α describes the only (infinitely repeated) execution step of the horizontal primitive, say Prim, of the first connector in Figure 1. In that execution step, Prim 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 $\{\mathsf{A}\,,\mathsf{C}\,,\mathsf{E}\}$-transition of ϵ means that the left-hand primitive of the fourth connector in Figure 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).

If α denotes a PA, let $\mathsf{State}(\alpha)$, $\mathsf{Port}(\alpha)$, and $\mathsf{init}(\alpha)$ denote its states, ports, and initial state (see [13, Appendix A] for formal definitions).

We adopt strong bisimilarity on PA as behavioral equivalence [12]: if α and β are bisimilar, denoted by $\alpha \approx \beta$, α can "simulate" every transition of β in every state and vice versa (see [13, Appendix A] for a formal definition).

Individual PA describe the behavior of individual connectors; the application of the existing product operator to such PA models connector composition [12]. We define this operator in two steps.[3] 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 common 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* $\wp(\mathbb{P}\textsc{ort})^2 \times \wp(\mathbb{P}\textsc{ort})^2$ *defined as:*

$$(\mathcal{P}_\alpha\,, P_\alpha) \Diamond (\mathcal{P}_\beta\,, P_\beta) \ \textbf{ iff } \begin{bmatrix} P_\alpha \subseteq \mathcal{P}_\alpha \ \textbf{ and } \ P_\beta \subseteq \mathcal{P}_\beta \\ \textbf{and } \ \mathcal{P}_\alpha \cap P_\beta = \mathcal{P}_\beta \cap P_\alpha \end{bmatrix}$$

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

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

[3] This simplifies relating this product operator to the product operator of Section 3.

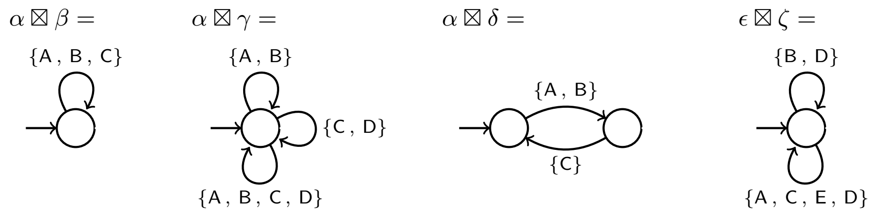

Fig. 3. Port automata describing the behavior of the example connectors in Figure 1, constructed using ⊠ (α, β, γ, δ, ϵ, and ζ denote the PA in Figure 2).

Definition 3 (Product operator). *The product operator, denoted by* _⊠_*, 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:*

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

$$\frac{\begin{array}{c} q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha \ \textbf{ and } \ q_\beta \in Q_\beta \\ \textbf{and } \ P_\alpha \cap \mathsf{Port}(\beta) = \emptyset \end{array}}{(q_\alpha\,,\, q_\beta) \xrightarrow{P_\alpha} (q'_\alpha\,,\, q_\beta)} \ (\textsc{Indep}\text{A}) \qquad \frac{\begin{array}{c} q_\beta \xrightarrow{P_\beta}_\beta q'_\beta \ \textbf{ and } \ q_\alpha \in Q_\alpha \\ \textbf{and } \ P_\beta \cap \mathsf{Port}(\alpha) = \emptyset \end{array}}{(q_\alpha\,,\, q_\beta) \xrightarrow{P_\beta} (q_\alpha\,,\, q'_\beta)} \ (\textsc{Indep}\text{B})$$

The previous definition reformulates the product of PA in [12], which is a simplification of the product of CA in [10]. Figure 3 shows examples of the application of ⊠. The $\{\mathsf{A}\,,\, \mathsf{B}\,,\, \mathsf{C}\,,\, \mathsf{D}\}$-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 (true concurrency). The following lemma states that bisimilarity is a congruence. See [12, Theorem 1] for a proof.

Lemma 1. $\big[\alpha \approx \beta$ **and** $\gamma \approx \delta\big]$ **implies** $\alpha \boxtimes \gamma \approx \beta \boxtimes \delta$

Furthermore, ⊠ is associative and commutative.

Interestingly, ⊠ *"transitively" propagates synchrony* over successive applications. We explain what this means with an example. Suppose Alice knows about ports $\{\mathsf{A}\,,\, \mathsf{B}\}$ and has one transition in which she fires exactly those ports. Similarly, suppose Bob knows about ports $\{\mathsf{B}\,,\, \mathsf{C}\}$ and has one transition in which he fires exactly those ports. Because these two transitions satisfy $\between$, the product of Alice and Bob has one transition labeled by $\{\mathsf{A}\,,\, \mathsf{B}\,,\, \mathsf{C}\}$. This means that Alice and Bob always synchronize on their common port B: Alice can perform her transition (i.e., is *willing* to fire B) only if Bob can perform his (i.e., is *ready* to fire B) and vice versa. Now, suppose Carol knows about ports $\{\mathsf{C}\,,\, \mathsf{D}\}$ and has one transition in which she fires exactly those ports. By the same reasoning as before, the product of [the product of Alice and Bob][4] 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.[5]

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 Dave knows about ports {E , F} and has one transition in which he fires exactly those ports. The product of Alice, Bob, Carol, and Dave computed by a tool using the centralized approach has three transitions: one labeled by {A , B , C , D} (Alice, Bob, Carol make a transition), another labeled by {E , F} (Dave makes a transition), and yet another labeled by {A , B , C , D , E , F} (Alice, Bob, Carol and Dave coincidentally make a transition at the same time by true concurrency). At run-time, in every iteration of its main loop, the thread simulating this big automaton nondeterministically picks one of those transitions, checks it for enabledness (in which case all ports are ready to fire), and if so, executes it. By this scheme, as soon as the automaton thread has selected the transition labeled by {A , B , C , D}, the transition labeled by {E , F} has to wait for the next iteration, even if it is enabled already in the current iteration. In other words, Dave cannot execute at its 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 Dave individually. The problem with the distributed approach is the communication necessary for computing $\boxtimes$ at run-time. To see this, suppose that we indeed have separate threads simulating the automata of Alice, Bob, Carol, and Dave. Now, if Alice at some point becomes willing to execute her {A , B} transition, she must ask Bob if he is ready to execute his {B , C} transition. Before he can answer Alice's question, however, Bob in turn must ask Carol if she is ready to execute her {C , D} transition. All this communication negatively affects throughput: it takes much longer for Alice, Bob, and Carol to agree on synchronously executing their individual transitions than for one big automaton to make and carry out such a decision by itself. Nevertheless, the distributed approach enhances parallelism: Dave can execute his transition *while* Alice, Bob, and Carol communicate to come to an agreement.

3 A New Local Product Operator

The approaches of the previous section force one to choose between two desirable properties: high throughput between *inter*dependent port automata (PA), at the cost of parallelism, and maximal parallelism between *in*dependent ones, at the cost of throughput. We need to find a middle ground between the purely centralized and fully distributed approaches that has both these desirable qualities.

[4] Square brackets for readability.

[5] This property of $\boxtimes$ models an important feature of Reo: compositional construction of globally synchronous protocol steps out of locally synchronous parts.

Working toward such an approach, we start from the purely distributed approach of computing $\boxtimes$ at run-time through global, transitive communication between automaton threads (e.g., Alice talks to Bob, who in turn talks to Carol, etc.). The idea is to bound this transitivity: generally, when some Alice asks some Bob if he is ready to fire a transition involving common ports, Bob *should* immediately answer *without engaging others*. By doing so, Alice and Bob achieve a higher throughput, while independent others can still execute at their own pace.

In the proposed approach, automaton threads no longer compute $\boxtimes$: instead, they compute a new product operator whose run-time computation requires only local communication. Problematically, however, computing 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 firing of Alice's ports (including her ports common 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 common ports. In the rest of this section, we formalize the new product operator and make a first effort at studying under which circumstances substituting $\boxtimes$ with the new product operator 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 $\lozenge$). In that case, they agree on which of their common 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:*

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

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*, and:*

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

Figure 4 shows examples of the application of $\boxdot$. The following lemma states that bisimilarity is a congruence. See [13, Appendix D] for a proof.

Lemma 2. $\big[\alpha \approx \beta$ **and** $\gamma \approx \delta\big]$ **implies** $\alpha \boxdot \gamma \approx \beta \boxdot \delta$

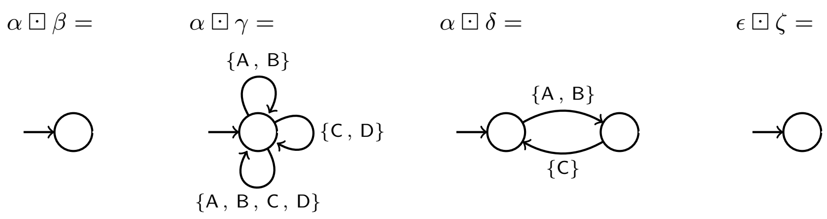

Fig. 4. Port automata constructed using $\boxdot$ (α, β, γ, δ, ϵ, and ζ denote the PA in Figure 2).

Furthermore, $\boxdot$ is commutative but generally *not* associative. This makes using $\boxdot$ for modeling purposes nontrivial. We address this issue in Section 5. To minimize numbers of parentheses in our notation, we assume right-associativity for $\boxdot$. For instance, we write $\alpha \boxdot \beta \boxdot \gamma \boxdot \delta$ for $\alpha \boxdot (\beta \boxdot (\gamma \boxdot \delta))$.

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 Figure 4 (cf. the two products in the middle of Figure 3) but not for the l-products on the sides. To determine when substituting $\boxtimes$ with $\boxdot$ is semantics-preserving, we first 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 $\mathbb{P}\mathrm{A} \times \mathbb{P}\mathrm{A}$ 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 1. $\big[\alpha \sqsubseteq \beta$ **and** $\beta \sqsubseteq \alpha\big]$ **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 common ports that Bob fires or vice versa, Alice and Bob agree on their common ports. See [13, Appendix D] for a proof.

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

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. See [13, Appendix D] for a proof (which uses Lemma 3).

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

The product of Alice and Bob is not necessarily a subautomaton of their l-product: if Alice and Bob agree on which of their common 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 common ports to fire implies their strong agreement.

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

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

The next lemma states that if Alice and Bob conditionally strongly agree, their product is a subautomaton of their l-product (cf. Lemma 4). See [13, Appendix D] for a proof.

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

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). See [13, Appendix D] for a proof (which uses Proposition 1 and Lemmas 4, 5).

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

4 Substituting $\boxtimes$ with $\boxdot$, a 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 $\lozenge\!\!\!\blacklozenge$-based characterization, although (conjectured to be) complete, hard to apply in practice. In this section, we therefore study a cheaper characterization of (a subset of) conditionally strongly agreeing port automata (PA) without restricting the applicability of $\boxdot$ for our present purpose.

In Section 2, we explained reduction of parallelism in terms of independent PA. Therefore, substituting $\boxtimes$ with $\boxdot$ should be semantics-preserving at least when applied to such PA. We start by formally defining when Alice and Bob are independent: in that case, they have no common 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:*

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

The next lemma states that if Alice and Bob are independent, they conditionally strongly agree (because their independence means that Alice and Bob have no common ports). See [13, Appendix D] for a proof.

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

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: $\mathcal{O}(1)$ versus $\mathcal{O}(n_1 n_2)$. The next lemma states another important property, namely that $\boxdot$ preserves independence: if Alice is independent of Bob and Carol individually, she is independent of Bob and Carol together. See [13, Appendix D] for a proof.

Lemma 7. $\big[\alpha \asymp \beta$ **and** $\alpha \asymp \gamma\big]$ **implies** $\alpha \asymp \beta \boxdot \gamma$

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 satisified by the PA encountered in code generation of a composite system. 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 common with Alice, involves *only* ports common with Alice. In other words, if common 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.

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:*

$$\begin{array}{l}(Q_\beta\,,\,\mathcal{P}_\beta\,,\,\longrightarrow_\beta\,,\,\imath_\beta)\\ \mapsto \alpha\end{array} \ \textbf{iff}\ \left[\begin{array}{c}\left[\left[\begin{array}{l} q_\beta \xrightarrow{P_\beta} q'_\beta \ \textbf{and} \\ P_\beta \cap \mathsf{Port}(\alpha) \neq \emptyset \end{array}\right] \ \textbf{implies}\ P_\beta \subseteq \mathsf{Port}(\alpha)\right] \\[1ex] \textbf{for all}\ q_\beta\,,\,q'_\beta\,,\,P_\beta \end{array}\right]$$

The next lemma states that if Bob is a slave of Alice, they conditionally strongly agree (i.e., Alice forces her will upon Bob). See [13, Appendix D] for a proof.

Lemma 8. $\beta \mapsto \alpha$ **implies** $\beta \mathrel{\langle\!\blacklozenge\!\rangle} \alpha$

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: $\mathcal{O}(n_1)$ versus $\mathcal{O}(n_1 n_2)$. The next lemma states another important property, namely that $\boxdot$ preserves slavery: if Bob is a slave of Alice, he is a slave of Alice and Carol together. See [13, Appendix D] for a proof.

Lemma 9. $\beta \mapsto \alpha$ **implies** $\beta \mapsto \alpha \boxdot \gamma$

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:*

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

The next 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). See [13, Appendix D] for a proof (which uses Lemmas 6, 8).

Lemma 10. $\beta \Mapsto \alpha$ **implies** $\beta \mathrel{\blacklozenge} \alpha$

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 next lemma states another important property, namely that $\boxdot$ preserves conditional slavery: if Bob is a conditional slave of Alice and Carol individually, he is a conditional slave of Alice and Carol together. The corollary following this lemma generalizes this result from 2 to k individuals. See [13, Appendix D] for a proof (which uses Lemmas 7, 9).

Lemma 11. $\big[\beta \Mapsto \alpha$ **and** $\beta \Mapsto \gamma\big]$ **implies** $\beta \Mapsto \alpha \boxdot \gamma$

Corollary 1. $\big[\beta \Mapsto \alpha_1$ **and** $\cdots$ **and** $\beta \Mapsto \alpha_k\big]$ **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. 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 $\boxdot$ is right-associative).

Definition 11. $\mathcal{A}$ *denotes the smallest set induced by the following rule:*

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

Strictly, $\mathcal{A}$ contains terms over $(\mathbb{P}\text{A}, \boxdot)$, which represent PA, rather than actual elements from $\mathbb{P}\text{A}$. Nevertheless, we often call the elements from $\mathcal{A}$ "PA" for simplicity. Also, instead of writing $\alpha_1 \boxdot \cdots \boxdot \alpha_k$, we sometimes write $\alpha_1 \cdots \alpha_k$.

The following theorem states that for every PA in $\mathcal{A}$, substituting $\boxtimes$ for $\boxdot$ is semantics-preserving. See [13, Appendix D] for a proof (which uses Lemma 10 and Corollary 1).

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

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 considerd in this paper. For instance, Figures 3, 4 show that $\beta \boxdot \delta = \beta \boxtimes \delta$ (Figure 2 defines β and δ). By the commutativity of $\boxdot$ and $\boxtimes$, we have also $\delta \boxdot \beta = \delta \boxtimes \beta$. 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 common with β (only B), it fires no other ports (β, in contrast, does fire another port in that case, namely C).

Previously, we claimed that the subclass of PA characterized in this section (i.e., $\mathcal{A}$ in Definition 11) does not restrict the applicability of $\boxdot$ for our purpose. 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 2. $\mathcal{B} \subseteq \mathcal{A}$

The combination of Proposition 2 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.

The previous interpretation of masters and slaves corresponds exactly to the notion of synchronous and asynchronous *regions* in the Reo literature [5,8]. Roughly, one can always split a connector into subconnectors—the regions—such that firings of ports in such a subconnector are either purely independent (i.e., always, only one port fires at a time) or require some synchronization (i.e., at least once, more than one port fires). Furthermore, the synchronous regions of a connector are maximal in the sense that no two synchronous regions have common ports: all synchronous regions are, by definition, pairwise independent. Consequently, the PA describing the l synchronous regions of a connector can act as the l masters in a PA term from $\mathcal{B}$.

To actually obtain those PA, for every synchronous region, a code generator during compilation computes the *existing* product of the PA describing the constituents of that particular region (finding the synchronous regions of a connector is trivial). At compile-time, this resembles the purely centralized approach, while at run-time, it ensures high throughput between interdependent "small" PA for constituents of the same synchronous region (i.e., no run-time computation of product operators within synchronous regions). The asynchronous regions then form the “glue” between the synchronous regions: the PA for every asynchronous region has the same shape as δ in Figure 2,[6] and consequently, they can act as the k conditional slaves in a PA term from $\mathcal{B}$. Finally, at run-time, the automaton threads executing the generated code compute the l-product operators.

In summary: a code generator can always process the set of PA describing a connector to a form that satisfies $\mathcal{B}$, by computing $\boxtimes$ between interdependent PA belonging to the same synchronous region at compile-time (for the sake of throughput), and by computing $\boxdot$ between the resulting “medium” PA plus the PA for the asynchronous regions at run-time (for the sake of parallelism). Proposition 2 and Theorem 2 ensure that this is semantics-preserving.

5 Note on Associativity

The associativity of $\boxtimes$ plays a role in the centralized approach and is even more important in the distributed approach. In the centralized approach, it guarantees that it does not matter in which particular order a code generator computes the product of the port automata (PA) for the constituents of a connector—all have the same semantics. In the distributed approach, it guarantees that it does not matter in which order PA threads communicate with each other: the PA term corresponding to a particular communication order is always bisimilar to the original (because one can freely move parentheses).

Now, recall from Section 3 that $\boxdot$ is generally *not* associative. The structure of the PA terms from $\mathcal{A}$ also reflects this (and the proof of Theorem 2 exploits this structure). This means that the PA constituting such terms must communicate in a particular order at run-time for the substitution of $\boxtimes$ with $\boxdot$ to be semantics-preserving. This can kill performance and seems a serious practical problem. For reasons of space, we postpone a full exposition of our solution to this problem to a future paper; interested readers may consult [13].

6 Related Work and Conclusion

Related work. 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 [7,8,9]. Furthermore, Clarke and

[6] Port automaton δ in Figure 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 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 that PA can act as a conditional slave in a $\mathcal{B}$-term.

Proença explored connector splitting in the context of the connector coloring semantics [3]. 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 [5]. Although, as shown in Section 4, 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 [6].)

Also related to the work presented in this paper is the work of Kokash et al. on *action constraint automata* (ACA) [14]. 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.

Conclusion. Existing approaches to implementing connectors force one to make a choice between high throughput (at the cost of parallelism) and maximal 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 code generation for connectors), one can use this new operator instead of the existing one to get both high throughput and maximal parallelism in a semantics-preserving way.

Although we developed our results for PA, they generalize straightforwardly to the more powerful constraint automata (CA) [10]. See [13] for more details.

While inspired by Reo, our results apply to every language whose programs can be described by automata satisfying the characterizations in Section 4. For instance, a possible application of our results outside Reo is *projection* in choreography languages [15,16,17,18,19,20]. See [13] for more details.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. MSCS 14(3), 329–366 (2004)
2. Arbab, F.: Puff, The Magic Protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011)

3. Clarke, D., Proença, J.: Partial Connector Colouring. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 59–73. Springer, Heidelberg (2012)
4. Jongmans, S.S., Arbab, F.: Modularizing and Specifying Protocols among Threads. In: Proceedings of PLACES 2012. EPTCS. CoRR, vol. 109, pp. 34–45 (2013)
5. Jongmans, S.S., Clarke, D., Proença, J.: A Procedure for Splitting Processes and its Application to Coordination. In: Proceedings of FOCLASA 2012. EPTCS. CoRR, vol. 91, pp. 79–96 (2012)
6. Jongmans, S.S., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic Code Generation for the Orchestration of Web Services with Reo. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 1–16. Springer, Heidelberg (2012)
7. Proença, J., Clarke, D., De Vink, E., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: Proceedings of SAC 2012, pp. 1510–1515. ACM (2012)
8. Proença, J.: Synchronous Coordination of Distributed Components. PhD thesis, Leiden University (2011)
9. Proença, J., Clarke, D., De Vink, E., Arbab, F.: Decoupled execution of synchronous coordination models via behavioural automata. In: Proceedings of FOCLASA 2011. EPTCS. CoRR, vol. 58, pp. 65–79 (2011)
10. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. SCP 61(2), 75–113 (2006)
11. Jongmans, S.S., Arbab, F.: Overview of Thirty Semantic Formalisms for Reo. SACS 22(1), 201–251 (2012)
12. Koehler, C., Clarke, D.: Decomposing Port Automata. In: Proceedings of SAC 2009, pp. 1369–1373. ACM (2009)
13. Jongmans, S.S., Arbab, F.: Global Consensus through Local Synchronization (Technical Report). Technical Report FM-1303, CWI (2013)
14. Kokash, N., Changizi, B., Arbab, F.: A Semantic Model for Service Composition with Coordination Time Delays. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 106–121. Springer, Heidelberg (2010)
15. Bravetti, M., Zavattaro, G.: Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
16. Bravetti, M., Zavattaro, G.: Contract Compliance and Choreography Conformance in the Presence of Message Queues. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 37–45. Springer, Heidelberg (2009)
17. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. TCS 328(1-2), 19–37 (2004)
18. Fu, X., Bultan, T., Su, J.: Realizability of Conversation Protocols with Message Contents. IJWSR 2(4), 68–93 (2005)
19. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centered Programming for Web Services. TOPLAS 34(2), 8:1–8:78 (2012)
20. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: Proceedings of POPL 2008, pp. 273–284. ACM (2008)