SOCA (2015) 9:311–339
DOI 10.1007/s11761-015-0177-y

SPECIAL ISSUE PAPER

Partially distributed coordination with Reo and constraint automata

Sung-Shik T. Q. Jongmans · Francesco Santini · Farhad Arbab

Received: 27 February 2014 / Revised: 25 March 2015 / Accepted: 31 March 2015 / Published online: 22 April 2015

Abstract Coordination languages, such as Reo, have emerged for the specification and implementation of interaction protocols among concurrent entities, manifested as connectors. In this paper, we describe a theoretical justification and a practical proof-of-concept tool for automatically generating partially distributed, partially centralized implementations of Reo connectors. Such implementations have three performance advantages: faster compilation at build time (compared to a purely centralized approach), reduced latency at run time (compared to a purely distributed approach), and improved parallelism at run time (compared to a purely centralized approach). Our theory relies on the definition of a new product operator on constraint automata (Reo's formal semantics), which we use to formally justify distributions of disjoint parts of a coordination scheme over different machines according to several possible motivations (e.g., performance, QoS constraints, privacy, resource availability, and network topology). To exemplify our work, in a case study, we show and explain how a generated connector implementation can be executed.

Francesco Santini: This work was carried out during the second author's tenure of the ERCIM "Alain Bensoussan" Fellowship Programme. This programme is supported by the Marie Curie Co-funding of Regional, National, and International Programmes (COFUND) of the European Commission.

S.-S. T. Q. Jongmans (✉) · F. Arbab
Centrum Wiskunde & Informatica (CWI), Science Park 123, Amsterdam, The Netherlands
e-mail: jongmans@cwi.nl

F. Arbab
e-mail: farhad@cwi.nl

F. Santini
Istituto di Informatica e Telematica (CNR), Via Moruzzi 1, Pisa, Italy
e-mail: francesco.santini@iit.cnr.it

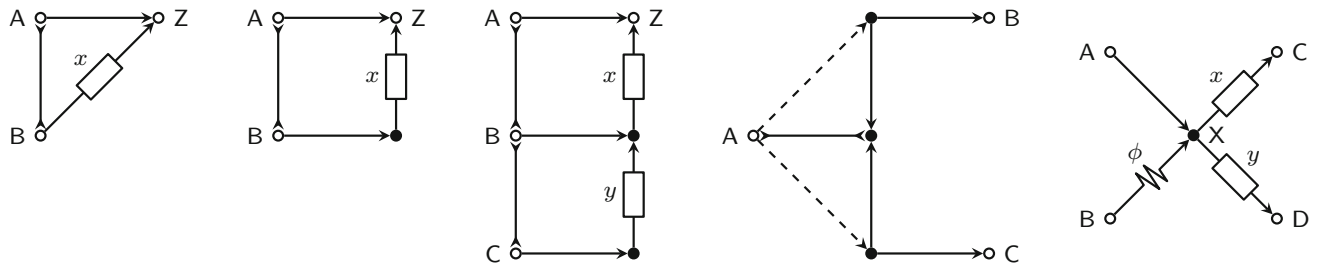
Keywords Reo coordination language · Distributed computation · Web service composition · Orchestration

1 Introduction

1.1 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 language for compositional construction of *connectors*: communication media through which entities can interact with each other. Figure 1 shows (classic, except Fig. 1e) example connectors in a 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), developers can construct arbitrarily complex connectors. While nodes have fixed dataflow behavior, Reo features an open-ended set of channels: developers can define their own channels with custom semantics.

Reo has several software engineering advantages as a domain-specific language for programming interaction protocols [31]. For instance, the use of Reo forces developers to separate their computation code from their protocol code instead of intermixing computations with protocols (as usually done when both are implemented in the same language). This separation facilitates verbatim reuse, independent modification, and compositional construction of protocol implementations (i.e., Reo connectors) in a straightforward way. Moreover, Reo has a formal foundation, which enables formal analysis of connectors (e.g., model checking [41]). This makes statically verifying that a given protocol does not dead-



(a) Alternator **(b)** Alternator$_2$ **(c)** Alternator$_3$ **(d)** ExclRouter **(e)** ExampleConn

Fig. 1 Example connectors: members of the Alternator family and ExampleConn

lock, for instance, relatively easy. Such protocol analyses are much harder to perform on lower-level general-purpose languages, especially when computations and protocols are intermixed.

Reo has been successfully used as a language for expressing *orchestration protocols* for composition of Web services [37,38]. The previous software engineering advantages of Reo aside, one of the main differences between Reo and BPEL [53], generally considered the most important orchestration language in industry [8,55], is that Reo is declarative [38]: in Reo, one specifies what—instead of how—interaction among services takes place. Given the success of declarative programming (e.g., logic programming and functional programming) in many areas of computer science, the availability of a declarative language for orchestrating services has value: developers accustomed to declarative programming may prefer Reo's style over BPEL's imperative style.

1.2 Problem

To use Reo connectors 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 [32]. In the *distributed approach*, one implements the behavior of each of the k constituents of a connector (its nodes and channels) 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. The distributed approach has the advantage of fast compilation at build time and high parallelism at run time. However, this comes at the cost of higher latency at run time (because of communication necessary for executing distributed algorithms). In contrast, the centralized approach has the advantage of low latency at run time but at the cost of slow compilation and reduced parallelism. Moreover, the amount of code generated in the centralized approach may be exponential in k, in which case the output is prohibitively big and the time to produce it prohibitively long. Proença et al. [59,60] observe that a partially distributed *hybrid approach*, where parts of a connector are compiled according to the centralized approach and deployed in a distributed fashion, is generally ideal: a hybrid approach strikes a middle ground between latency and parallelism at run time while achieving reasonably fast compilation at build time.

In this paper, we address the problem that no systematic, formally justified scheme exists to automatically construct connector implementations according to the hybrid approach. We do so in the context of *constraint automata* (CA) [4], a formalism, based on finite automata on infinite words, that is used to define the semantics of Reo. Essentially, the states of a CA model the internal configurations of a connector, while its transitions model that connector's atomic execution steps. We perform all our theoretical work in this paper at the level of CA rather than at the level of Reo. As such, our results are generally applicable to any coordination, choreography, or orchestration langauge whose semantics maps into CA. Our current tools, however, support Reo, whose compiler is responsible for translating Reo into CA, in a manner transparent to developers.

Observe that the issues involving centralized versus distributed control of distributed applications are *not* specific to Reo; they are inherent in distributed applications and arise explicitly or implicitly, in any implementation using any language. Striking a balance between the two extremes in a hybrid approach, therefore, solves a generic problem, which in our work, we tackle with the syntax of Reo and its CA semantics.

1.3 Contribution

We use Reo's notion of *(a)synchronous regions* and an extended recent result on connector composition to automatically construct formally sound hybrid connector implementations, using CA as our semantic formalism. We implement this construction on top of an existing Reo-to-Java centralized-code generator [31]. This enables a case study on *distrib-*

uted service orchestration by reusing an existing Reo-based orchestration framework [37]. This case study shows that the hybrid approach can improve the centralized approach not only in terms of run-time parallelism but also latency.

Observe that the notions of synchronous and asynchronous regions are *not* mere artifacts of using Reo as our language to express coordination of distributed applications. These regions naturally arise in distributed applications, and the choice of Reo simply makes them explicit.

The value of our theoretical results extends beyond being an essential contribution to code generation technology for Reo, in two ways. First, because we formulate our results generally in terms of constraint automata (i.e., Reo's semantics, essentially independent of Reo), any system expressible in terms of those automata can benefit from these results. This enables algorithmically generating hybrid implementations of systems specified in other languages. 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 work by Lynch [43].

We organize this paper as follows. In Sect. 2, we give a concise overview of Reo and constraint automata. In Sect. 3, we explain the formal theory behind our hybrid connector implementations and how to automatically generate them. In Sect. 4, we discuss the salient aspects of our implementation. In Sect. 5, we exemplify our code generator with a distributed service orchestration scenario. In Sect. 6, we compare the performance of centralized and hybrid connector implementations. In Sect. 7, we discuss related work. In Sect. 8, we discuss other possible applications of the work presented in this paper. Finally, Sect. 9 concludes this paper.

This paper extends the work in [36] with additional formal definitions and proof outlines, an improved comparison with related work, new examples, and an improved exposition of our case study. Full technical details and proofs appear in a technical report [35].

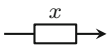
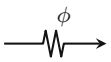
2 Reo and constraint automata

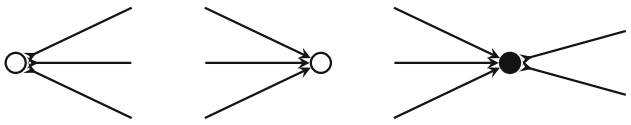
In this section, we discuss preliminaries on Reo and its constraint automaton semantics.

2.1 Reo

Reo is a language for compositional construction of concurrency protocols, manifested as connectors [1,2]. Connectors consist of channels and nodes, organized in a graph-like structure. Every channel consists of two ends and a constraint that relates the timing and the contents of the data flows at those ends. A channel end has one of two types: *source ends* accept data (i.e., a source end of a channel connects to that channel's data source/producer) and *sink ends* dispense data (i.e., a sink end of a channel connects to that channel's data sink/consumer). Reo makes no other assumptions about channels. This means, for instance, that channels in Reo may have two source ends. Table 1 shows the syntax and an informal description of five common channels.

Table 1 Syntax and semantics of common channels

Name	*Graphical syntax*	*Semantics*
sync		Atomically takes a data item d from its source end and writes d to its sink end
lossysync-nd		Atomically takes a data item d from its source end and nondeterministically either writes d to its sink end or loses d
syncdrain		Atomically takes data items from both its source ends and loses them
fifo		Takes a data item d from its source end, then stores it in a private memory cell x, then writes d to its sink end
filter		Atomically takes a data item d from its source end and writes d to its sink end (if $d \in \phi$) or loses d (otherwise)



(a) Source **(b)** Sink **(c)** Mixed

Fig. 2 Node types

Entities communicating through a connector perform I/O operations—writes and takes—on its nodes. Reo has three types of nodes: *source nodes* on which only source ends coincide (see Fig. 2a), *sink nodes* on which only sink ends coincide (see Fig. 2b), and *mixed nodes* on which both types of channel end coincide (see Fig. 2c). Informally, nodes behave as follows.

– A source node n has *replicator semantics*. Once a communicating entity attempts to write a piece of data d on n, this node first suspends that operation. Subsequently, n notifies the channels whose source ends coincide on n that it offers d. Once each of those channels has notified n that it accepts d, n resolves the write: the write operation terminates successfully and atomically, n dispenses

(a copy of) d to each of its coincident source ends. Source nodes forbid takes.
- A sink node n has *merger semantics*. Once a communicating entity attempts to take a piece of data from n, this node first suspends that operation. Subsequently, n notifies the channels whose sink ends coincide on n that it accepts a piece of data. Once at least one of these channels has notified n that it offers a piece of data d, n resolves the take: atomically, n fetches d from the appropriate channel end and dispenses it to the entity attempting to take. If multiple sink ends offer a data item, n chooses one of them nondeterministically. Sink nodes forbid writes.
- A mixed node executes both the merge/replicate behavior as discussed above. Mixed nodes forbid I/O.

Importantly, nodes cannot generate or lose data, nor can they store data between execution steps: once a piece of data enters a node through a coincident channel's sink ends, (copies of) that data must immediately exit the node through its coincident channels' source ends, in the same step. As such, nodes synchronize their coincident channel ends. This behavior of nodes means that synchronization and exclusion in Reo propagate through sub-circuits and are preserved by composition.

So far, we explicitly distinguished between three language constructs in Reo: channels, nodes, and connectors. In the rest of this paper, for simplicity (but without loss of generality), we represent every channel c as its *corresponding connector*, which consists of two nodes connected by c. To define the semantics of c, it suffices to define the semantics of its corresponding connector. Furthermore, we assume that at most one source end and at most one sink end coincide on every node [4]. We model such "binary nodes" with *ports*. Ports occur either on the boundary between a connector and its environment or inside a connector. If a connector consists only of boundary ports, we call it a *primitive*; otherwise, we call it a *composite*. To simulate the semantics of nodes with more than two coincident channel ends, we use two ternary primitives: Merger and Replicator. These primitives can compose into a Node composite, which connects $n \geq 0$ input ports to $m \geq 0$ output ports and behaves as a node on which n source ends and m sink ends coincide (where $n + m > 0$). Figure 3 shows an example.

For every connector with "multiary nodes," there exists a behaviorally equivalent connector with only binary nodes. Although such an equivalent connector has more primitives than its original, formally modeling behavior and composition becomes easier with only binary nodes. After all, connectors with only binary nodes eliminate the need of modeling the merge/replicate behavior of nodes as first-class concepts. Instead, formal models of the added Merger and Replicator primitives represent merge/replicate behavior in the same way as formal models of channels represent channel behavior. The assumption of only binary nodes thus allows us to keep later definitions in this paper—Definitions 1, 4, and 5 in particular—clean as compared to definitions in which merge/replicate behavior have first-class status.

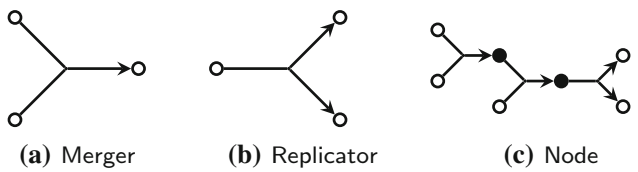


Fig. 3 Merger, Replicator, and Node for $n, m = 3, 2$

2.2 Constraint automata

Many formalisms exist for mathematically defining the semantics of connectors [28]. In this paper, we adopt the same formalism as the existing code generator that we use: constraint automata (CA) [4]. A CA consists of finite sets of states and transitions. States represent the internal configurations of a connector; transitions describe the atomic steps of the protocol specified by a connector. Formally, we represent a transition as a tuple of four elements: a source state, a *synchronization constraint*, a *data constraint*, and a target state. A synchronization constraint is a set that specifies on which ports a data item flows (i.e., which ports synchronize); a data constraint is a logical formula that specifies which particular data items flow on which of those ports.

Figure 4 shows example CA, where A and B refer to ports. Informally, the data constraint $d(\mathsf{A}) = d(\mathsf{B})$ means that the data item flowing on port A equals the data item flowing on port B; the data constraint $\top$ means that it does not matter

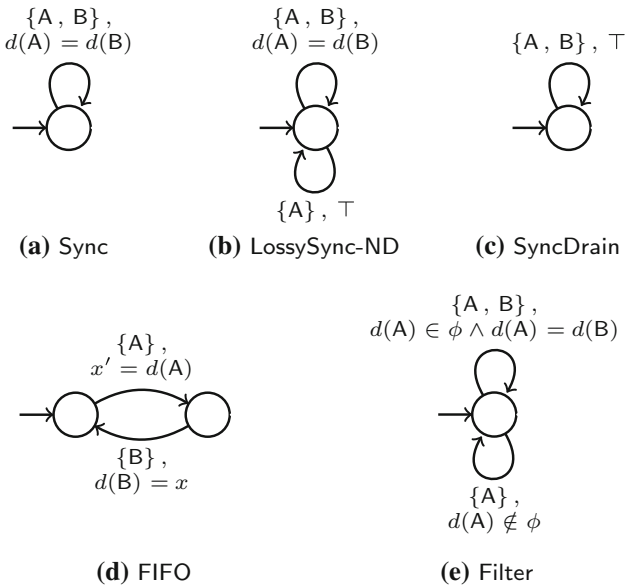


Fig. 4 Constraint automata for the channels in Table 1 (between ports A and B)

$$\begin{bmatrix} (Q_\alpha\,,\,\mathcal{P}_\alpha\,,\,\mathcal{M}_\alpha\,,\,\longrightarrow_\alpha\,,\,\imath_\alpha) \\ \preceq^R \\ (Q_\beta\,,\,\mathcal{P}_\beta\,,\,\mathcal{M}_\beta\,,\,\longrightarrow_\beta\,,\,\imath_\beta) \end{bmatrix} \textbf{ iff } \left[\begin{array}{c} R \subseteq Q_\alpha \times Q_\beta \textbf{ and } \mathcal{P}_\alpha = \mathcal{P}_\beta \textbf{ and } \imath_\alpha\ R\ \imath_\beta \textbf{ and} \\ \left[\left[\begin{array}{l} q_\alpha \xrightarrow{P,f_\alpha}_\alpha q'_\alpha \\ \textbf{and } q_\alpha\ R\ q_\beta \end{array}\right] \textbf{ implies } f_\alpha \Rightarrow \bigvee\left\{ f_\beta \;\middle|\; \left[q_\beta \xrightarrow[\textbf{for some } q'_\beta]{P,f_\beta}_\beta q'_\beta \textbf{ and } q'_\alpha\ R\ q'_\beta\right] \right\}\right] \\ \textbf{for all } f_\alpha\,,\,P\,,\,q_\alpha\,,\,q'_\alpha\,,\,q_\beta \end{array}\right]$$

$$\alpha \preceq \beta \qquad \textbf{iff } \left[\alpha \preceq^R \beta \textbf{ for some } R\right]$$

Fig. 5 Simulation preorder

which particular data items flow; the data constraint $x' = d(\mathsf{A})$ means that the value of x, a private (to the connector) memory cell, *after completing* the transition equals the data flowing on A *during* the transition (i.e., x is written to in the end); the data constraint $d(\mathsf{B}) = x$ means that the data flowing on B equals the value of private memory cell x *before* starting the transition (i.e., x is read from in the beginning). Finally, if ϕ represents a finite set of data items, the data constraint $d(\mathsf{A}) \in \phi$ abbreviates $\bigvee\{d(\mathsf{A}) = v \mid v \in \phi\}$. A data constraint f implies a data constraint g, denoted as $f \Rightarrow g$, if g admits at least the same distributions of data over ports and memory cells as f [4].

Let $\mathbb{S}\textsc{tate}$, $\mathbb{P}\textsc{ort}$, $\mathbb{M}\textsc{em}$, and $\mathbb{D}\textsc{c}$ denote universes of states, ports, memory cells, and data constraints, respectively.

Definition 1 The universe of CA, denoted by $\mathbb{C}\textsc{a}$, is the largest set of tuples $(Q, \mathcal{P}, \mathcal{M}, \longrightarrow, \imath)$ where:

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

If α denotes a CA, let $\mathsf{State}(\alpha)$, $\mathsf{Port}(\alpha)$, $\mathsf{Mem}(\alpha)$, and $\mathsf{init}(\alpha)$ denote its states, ports, (the names of its) memory cells, and initial state, respectively.

We adopt a behavioral equivalence on CA based on *bisimulation* [47]. Because transitions of CA have richer labels than those in standard labeled transition systems, simulation on CA has a more complex definition than usual [4].

Definition 2 The simulation preorder on CA, denoted by $\preceq \subseteq (\mathbb{C}\textsc{a} \times \wp(\mathbb{S}\textsc{tate}^2) \times \mathbb{C}\textsc{a}) \cup (\mathbb{C}\textsc{a} \times \mathbb{C}\textsc{a})$, is the relation defined in Fig. 5.

If $\alpha \preceq \beta$, we say that β simulates α. In that case, there exists a relation R on the sets of states of α and β such that, for all states q_α of α, if

- R relates q_α to a state q_β of β
- and α has a transition from q_α to a state q'_α involving ports P and distributing data according to data constraint f_α,

a transition from q_β to a state q'_β exists involving the same ports P and distributing data according to data constraint f_β in the same way (but f_β *may* admit also other data distributions forbidden by f_α). For instance, in Fig. 4, LossySync-ND and SyncDrain simulate Sync, but not vice versa: Sync cannot mimic the lossy transition of LossySync-ND, because Sync has no transition involving only port A. Similarly, although Sync has a transition involving ports A and B, it cannot mimic the transition of SyncDrain: $\top$ admits any assignment of data, while $d(\mathsf{A}) = d(\mathsf{B})$ admits only specific ones. We define behavioral equivalence in terms of the simulation preorder.

Definition 3 The behavioral equivalence on CA, denoted by $\approx \subseteq (\mathbb{C}\textsc{a} \times \wp(\mathbb{S}\textsc{tate}^2) \times \mathbb{C}\textsc{a}) \cup (\mathbb{C}\textsc{a} \times \mathbb{C}\textsc{a})$, is the relation defined as:

$$\alpha \approx^R \beta \quad \textbf{iff } \left[\alpha \preceq^R \beta \textbf{ and } \beta \preceq^{R^{-1}} \alpha\right]$$
$$\alpha \approx \beta \quad \textbf{iff } \left[\alpha \approx^R \beta \textbf{ for some } R\right]$$

We consider CA up to behavioral equivalence.

Individual CA describe the behavior of individual connectors; a *product operator* on CA models connector composition [4]. Its formal definition and a congruence theorem, proved by Baier et al. [4], follow below.

Definition 4 The product operator, denoted by $\_ \boxtimes \_$, is the operator on $\mathbb{C}\textsc{a}$ defined by the following equation:

$$\alpha \boxtimes \beta = \begin{pmatrix} \mathsf{State}(\alpha) \times \mathsf{State}(\beta), \mathsf{Port}(\alpha) \cup \mathsf{Port}(\beta), [] \\ \mathsf{Mem}(\alpha) \cup \mathsf{Mem}(\beta), \longrightarrow, (\mathsf{init}(\alpha), \mathsf{init}(\beta)) \end{pmatrix}$$

where $\longrightarrow$ denotes the smallest relation induced by Rule (1), Rule (3), and Rule (4) in Fig. 6.

Rule (1) states that whenever two automata each can fire a transition in which they *agree* on the involvement of shared ports, they can fire their transitions together in the composition (i.e., synchronously). The notion of agreement is formalized in the $\Diamond$ relation. Rule (3) states that whenever an automaton can fire a transition involving no shared ports, it can fire this transition any time it wants in the composition; Rule (4) is symmetric to Rule (3). Note that we do not use Rule (2) in Definition 4. Instead, we use and explain it in the next section.

Theorem 1 ($\approx$ is a congruence for $\boxtimes$)

$$\left[\alpha \approx \beta \textbf{ and } \gamma \approx \delta\right] \textbf{ implies } \alpha \boxtimes \gamma \approx \beta \boxtimes \delta$$

$$\frac{q_\alpha \xrightarrow{P_\alpha, f_\alpha}_\alpha q'_\alpha \ \textbf{and}\ q_\beta \xrightarrow{P_\beta, f_\beta}_\beta q'_\beta \ \textbf{and}\ (\mathsf{Port}(\alpha)\,,\, P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\, P_\beta)}{(q_\alpha\,,\, q_\beta) \xrightarrow{P_\alpha \cup P_\beta, f_\alpha \wedge f_\beta} (q'_\alpha\,,\, q'_\beta)} \quad (1)$$

$$\frac{q_\alpha \xrightarrow{P_\alpha, f_\alpha}_\alpha q'_\alpha \ \textbf{and}\ q_\beta \xrightarrow{P_\beta, f_\beta}_\beta q'_\beta \ \textbf{and}\ (\mathsf{Port}(\alpha)\,,\, P_\alpha) \blacklozenge (\mathsf{Port}(\beta)\,,\, P_\beta)}{(q_\alpha\,,\, q_\beta) \xrightarrow{P_\alpha \cup P_\beta, f_\alpha \wedge f_\beta} (q'_\alpha\,,\, q'_\beta)} \quad (2)$$

$$\frac{q_\alpha \xrightarrow{P_\alpha, f_\alpha}_\alpha q'_\alpha \ \textbf{and}\ q_\beta \in Q_\beta \ \textbf{and}\ P_\alpha \cap \mathsf{Port}(\beta) = \emptyset}{(q_\alpha\,,\, q_\beta) \xrightarrow{P_\alpha, f_\alpha} (q'_\alpha\,,\, q_\beta)} \quad (3)$$

$$\frac{q_\beta \xrightarrow{P_\beta, f_\beta}_\beta q'_\beta \ \textbf{and}\ q_\alpha \in Q_\alpha \ \textbf{and}\ P_\beta \cap \mathsf{Port}(\alpha) = \emptyset}{(q_\alpha\,,\, q_\beta) \xrightarrow{P_\beta, f_\beta} (q_\alpha\,,\, q'_\beta)} \quad (4)$$

(a) Rules

$$(\mathsf{Port}(\alpha)\,,\, P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\, P_\beta) \ \textbf{iff}\ \mathsf{Port}(\alpha) \cap P_\beta = \mathsf{Port}(\beta) \cap P_\alpha$$

$$(\mathsf{Port}(\alpha)\,,\, P_\alpha) \blacklozenge (\mathsf{Port}(\beta)\,,\, P_\beta) \ \textbf{iff}\ \begin{bmatrix} P_\alpha = \mathsf{Port}(\alpha) \cap P_\beta \ \textbf{or}\ P_\beta = \mathsf{Port}(\beta) \cap P_\alpha \\ \textbf{or}\ \ \mathsf{Port}(\alpha) \cap P_\beta = \emptyset = \mathsf{Port}(\beta) \cap P_\alpha \end{bmatrix}$$

(b) Auxiliary definitions

Fig. 6 Combining transitions

Henceforth, let $\bigboxtimes\{\alpha_1, \ldots, \alpha_k\}$ denote $\alpha_1 \boxtimes \cdots \boxtimes \alpha_k$. This is well defined, because $\boxtimes$ is associative and commutative.

3 Design: theoretical justification

3.1 Hybrid connector implementations

We start with presenting our design of hybrid connector implementations; in Sect. 3.2, we discuss the design of a tool that automatically generates corresponding code.

To first better explain the different implementation approaches for connectors with CA as formal semantics, let $X = \{\alpha_1, \ldots, \alpha_k\}$ denote a set of "small" CA, each of which models one of the k primitive constituents of the connector Conn to implement. We associate with X an *interpretation*, denoted by $[\![X]\!]$, which models the composition of the constituents in X (i.e., the full behavior of Conn):

$$[\![X]\!] = \bigboxtimes X \quad (1)$$

Every implementation of Conn—be it distributed, centralized, or hybrid—must be behaviorally equivalent to $[\![X]\!]$.

In the distributed approach, one first writes code for every $\alpha \in X$ and then deploys those k CA-implementations in k parallel processing units (e.g., processes, threads, actors). To ensure that those CA-implementations are behaviorally equivalent to $[\![X]\!]$, at run time, they must communicate with each other to check which of their transitions can fire: Generally, the enabledness of a transition of one CA-implementation depends on the enabledness of transitions of other CA-implementations; an example follows shortly. In the distributed approach, essentially, the k CA-implementations compute the $\boxtimes$-operators among them dynamically at run time. In the centralized approach, in contrast, one first computes $\bigboxtimes X$ (i.e., the full behavior of Conn), then writes code for the resulting "big" CA, and finally deploys this single CA-implementation in a single processing unit. By its construction, this single CA-implementation is behaviorally equivalent to $[\![X]\!]$. Finally, with the hybrid approach, one first constructs a *partition* $\mathcal{A} = \{A_1, \ldots, A_\ell\}$ of X, then computes $\bigboxtimes A$ for every *part* $A \in \mathcal{A}$, then writes code for the resulting "medium" CA, and finally deploys those ℓ CA-implementations in ℓ concurrent processing units. We associate with $\mathcal{A}$ an interpretation, denoted by $[\![\mathcal{A}]\!]$, by straightforwardly lifting the interpretation of sets of CA:

$$[\![\mathcal{A}]\!] = [\![\{[\![A_1]\!], \ldots, [\![A_\ell]\!]\}]\!] \quad (2)$$

One can easily show that the interpretation of X equals the interpretation of any of its partitions.

Proposition 1 $[\![X]\!] = [\![\mathcal{A}]\!]$ *for all partitions* $\mathcal{A}$ *of* X.

Proof Let $\mathcal{A} = \{A_1, \ldots, A_\ell\}$. Let $A_i = \{\alpha_{i,1}, \ldots, \alpha_{i,k_i}\}$ for all $1 \le i \le \ell$.

$$\begin{aligned}
&[\![X]\!] \\
&= \{\text{definition of } [\![\cdot]\!] \text{ for sets (Equation 1)}\} \\
&\quad \bigboxtimes X \\
&= \{\mathcal{A} = \{A_1, \ldots, A_\ell\} \text{ is a partition of X}\} \\
&\quad \bigboxtimes A_1 \cup \cdots \cup A_\ell \\
&= \{\text{definition of } A_1, \ldots, A_\ell \text{ and } \cup\} \\
&\quad \bigboxtimes\{\alpha_{1,1}, \ldots, \alpha_{1,k_1}, \ldots, \alpha_{\ell,1}, \ldots, \alpha_{\ell,k_\ell}\} \\
&= \{\text{definition of } \bigboxtimes\} \\
&\quad \alpha_{1,1} \boxtimes \cdots \boxtimes \alpha_{1,k_1} \boxtimes \cdots \boxtimes \alpha_{\ell,1} \boxtimes \cdots \boxtimes \alpha_{\ell,k_\ell} \\
&= \{\text{associativity/commutativity of } \boxtimes\} \\
&\quad (\alpha_{1,1} \boxtimes \cdots \boxtimes \alpha_{1,k_1}) \boxtimes \cdots \boxtimes (\alpha_{\ell,1} \boxtimes \cdots \boxtimes \alpha_{\ell,k_\ell}) \\
&= \{\text{definition of } \bigboxtimes\} \\
&\quad \bigboxtimes\{\alpha_{1,1}, \ldots, \alpha_{1,k_1}\} \boxtimes \cdots \boxtimes \bigboxtimes\{\alpha_{\ell,1}, \ldots, \alpha_{\ell,k_\ell}\} \\
&= \{\text{definition of } A_1, \ldots, A_\ell\} \\
&\quad \bigboxtimes A_1 \boxtimes \cdots \boxtimes \bigboxtimes A_\ell \\
&= \{\text{definition of } [\![\cdot]\!] \text{ for sets (Equation 1)}\} \\
&\quad [\![A_1]\!] \boxtimes \cdots \boxtimes [\![A_\ell]\!] \\
&= \{\text{definition of } \bigboxtimes\} \\
&\quad \bigboxtimes\{[\![A_1]\!], \cdots, [\![A_\ell]\!]\} \\
&= \{\text{definition of } [\![\cdot]\!] \text{ for sets (Equation 1)}\} \\
&\quad [\![\{[\![A_1]\!], \cdots, [\![A_\ell]\!]\}]\!] \\
&= \{\text{definition of } [\![\cdot]\!] \text{ for sets of sets (Equation 2)}\} \\
&\quad [\![\{A_1, \ldots, A_\ell\}]\!] \\
&= \{\text{definition of } \mathcal{A}\} \\
&\quad [\![\mathcal{A}]\!]
\end{aligned}$$

□

Essentially, in the hybrid approach, a code generator computes the $\boxtimes$-operators embedded in the "inner" interpretations statically at build time (as in the centralized approach), while the ℓ CA-implementations compute the $\boxtimes$-operators embedded in the "outer" interpretation dynamically at run time (as in the distributed approach).

To carry out the second, third, and fourth steps of the hybrid approach, we can use existing techniques from the distributed/centralized approach. The current challenge, thus, lies in the first step: finding a *reasonable* partition of X in a potentially huge search space. Theoretically, the total number of partitions of a k-cardinality set equals the kth *Bell number*, denoted by B_k, which grows rapidly in k. For instance, the number of possible partitions for Alternator in Fig. 1a, which consists of only 6 constituents, is $B_6 = 203$.

At one extreme, if we put every $\alpha \in X$ in its own part (i.e., $\mathcal{A} = \{\{\alpha_1\}, \ldots, \{\alpha_k\}\}$), we get the distributed approach, whose communication overhead increases latency at run time. However, at the other extreme, if we put every $\alpha \in X$ in the same part (i.e., $\mathcal{A} = \{\{\alpha_1, \ldots, \alpha_k\}\}$), we get the centralized approach, which removes all parallelism. Generally, the hybrid approach should avoid both these properties. As a compromise, we therefore adopt the following guideline for constructing partitions:

> Put those CA whose implementations *would* require (time-)expensive communication at run time in the same part; separate those CA that *will* require only "cheap" or no communication in different parts.

Informally, first, a partition $\mathcal{A}$ constructed according to this guideline yields lower latency (cf. the distributed approach), because those CA responsible for most of the communication overhead—caused by expensive communication—are compiled into one CA-implementation at build time (i.e., they populate the same part in $\mathcal{A}$). Second, $\mathcal{A}$ yields improved parallelism (cf. the centralized approach), because those CA requiring only cheap or no communication are deployed in parallel processing units at run time (i.e., each of them populates its own part in $\mathcal{A}$).

To more carefully exemplify the meaning of "expensive" and "cheap," let Alice, Bob, and Carol be three CA-implementations. In particular, let Alice, Bob, and Carol implement three Sync primitives in sequence between ports A, B, C, and D. Suppose that Alice's input port A has a pending write operation coming from the environment and that she wants to fire her {A, B}-transition (see Fig. 4a). Operationally, to fire this transition, Alice must atomically take the data item written on A from that port and write it to B (see Table 1). However, to guarantee atomicity, she must first ascertain that Bob is in fact *ready* to take a data item from B. If not, the data item that Alice writes to B has nowhere to go, which Reo's semantics forbids (i.e., ports/nodes cannot buffer data). So, before Alice takes the data item from A, she first asks Bob if he is ready to take from B. But Bob cannot immediately answer that question: he, in turn, must first ask Carol if she is ready to take a data item from C. If not, the data item that Bob atomically takes from B and writes to C has nowhere to go. In this simple example, Carol can answer Bob's question without further *derivative* communication by locally checking if D has a pending take operation. Generally, however, the chain of derivative communication can be much longer (ignoring, for the moment, the possibility of loops, which can be resolved). This makes the *initial* communication between Alice and Bob potentially very expensive. In summary, if Bob initiates derivative communication to answer a question from Alice, we call communication between Alice and Bob "expensive" (and "cheap" otherwise).

To identify which CA require only cheap communication, we extend the *local product theory* from CA without data constraints to unrestricted CA [29]. The idea is to introduce a new product operator on CA, called *l-product*, which models connector composition with cheap communication. By subsequently analyzing under which conditions substituting the existing product, which models expensive communication, with our new l-product in Eq. 2 is sound (i.e., faithful to Reo's semantics)—this is not always the case—we can devise an algorithm for computing reasonable partitions according to our cheap/expensive guideline. The formal definition of l-product and a congruence theorem follow below.

Definition 5 The l-product (where "l" stands for "local"), denoted by $\_ \boxdot \_$, is the operator on $\mathbb{C}\mathrm{A}$ defined by the following equation:

$$\alpha \boxdot \beta = \begin{pmatrix} \mathsf{State}(\alpha) \times \mathsf{State}(\beta), \mathsf{Port}(\alpha) \cup \mathsf{Port}(\beta), [] \\ \mathsf{Mem}(\alpha) \cup \mathsf{Mem}(\beta), \longrightarrow, (\mathsf{init}(\alpha), \mathsf{init}(\beta)) \end{pmatrix}$$

where $\longrightarrow$ denotes the smallest relation induced by Rule (2), Rule (3), and Rule (4) in Fig. 6.

The difference with Definition 4 is that the new transition relation is induced by Rule (2) in Fig. 6 instead of by Rule (1). We explain this as follows. Condition $(\mathsf{Port}(\alpha), P_\alpha) \Diamond (\mathsf{Port}(\beta), P_\beta)$ in the premise in Rule (1) states that, if two transitions agree on their shared ports, they can fire together. These transitions may, however, involve any number of ports not shared between those transitions. Exactly this freedom causes expensive communication when computing $\boxtimes$ at run time (e.g., Alice asks Bob about their shared port, but Bob in his transition wants to involve a port not shared with Alice, for which he asks Carol). To avoid this, condition $(\mathsf{Port}(\alpha), P_\alpha) \blacklozenge (\mathsf{Port}(\beta), P_\beta)$ in Rule (2) restricts the involvement of unshared ports: either one of the transitions involves only shared ports, or both transitions involve only unshared ports. Thus, only cheap communication occurs or no communication at all.

Theorem 2 ($\approx$ is a congruence for $\boxdot$)

$$\big[\alpha \approx \beta \textbf{ and } \gamma \approx \delta\big] \textbf{ implies } \alpha \boxdot \gamma \approx \beta \boxdot \delta$$

Proof (Outline) First, we use the premise and Definition 3 of $\approx$ to conclude the existence of relations R_1, R_2 such that $\alpha \approx^{R_1} \beta$ and $\gamma \approx^{R_2} \delta$. By further expanding $\approx$, we conclude $\alpha \preceq^{R_1} \beta$ and $\beta \preceq^{R_1^{-1}} \alpha$ and $\gamma \preceq^{R_2} \delta$ and $\delta \preceq^{R_2^{-2}} \gamma$ for later use. Next, we define:

$$(q_\alpha, q_\beta)\ R\ (q_\gamma, q_\delta) \textbf{ iff } \big[q_\alpha\ R_1\ q_\beta \textbf{ and } q_\gamma\ R_2\ q_\delta\big]$$

From this definition, we conclude:

$$(q_\gamma, q_\delta)\ R^{-1}\ (q_\alpha, q_\beta) \textbf{ iff} \big[q_\beta\ R_1^{-1}\ q_\alpha \textbf{ and } q_\delta\ R_2^{-1}\ q_\gamma\big]$$

Now, using $\alpha \preceq^{R_1} \beta$ and $\gamma \preceq^{R_2} \delta$, we straightforwardly establish $\alpha \boxdot \gamma \preceq^{R} \beta \boxdot \delta$. Similarly, we establish $\beta \boxdot \delta \preceq^{R^{-1}} \alpha \boxdot \gamma$. Finally, we conclude the required result by applying the definition of $\approx$. □

In the rest of this subsection, we address the issue of determining when substituting the existing product with l-product is sound. With this analysis, we aim at finding concrete conditions for algorithmically deciding when CA should belong to the same part in a partition for that partition to count as reasonable (with respect to our cheap/expensive guideline).

First, by applying set theory, we observe that the ⧫-condition in Rule (2) implies the ◊-condition in Rule (1) (i.e., cheap communication is a special case of expensive communication). Consequently, every transition of $\alpha \boxdot \beta$ induced by Rule (2) necessarily exists in $\alpha \boxtimes \beta$, where it is induced by Rule (1). This means that $\alpha \boxtimes \beta$ can mimic every transition of $\alpha \boxdot \beta$ in every state. In fact, the transition relation of $\alpha \boxtimes \beta$ is a superset of the transition relation of $\alpha \boxdot \beta$, while $\alpha \boxtimes \beta$ and $\alpha \boxdot \beta$ have the same state space and initial state. These observations allow us to prove that $\alpha \boxtimes \beta$ simulates $\alpha \boxdot \beta$ for R the identity relation on the common state space $\mathsf{State}(\alpha) \times \mathsf{State}(\beta)$.

$$\alpha \boxdot \beta \preceq^{R} \alpha \boxtimes \beta \tag{3}$$

Unsurprisingly, the converse of the previous implication—◊ implies ⧫—generally does not hold (i.e., expensive communication is not a special case of cheap communication). We can, however, characterize a set of CA couples such that the converse does hold for all their pairs of transitions: for every couple, (α, β) in this set, all communication between α and β is *actually cheap*.

Definition 6 The actually-cheap relation, denoted by $\langle\!\blacklozenge \subseteq \mathbb{C}\mathrm{A} \times \mathbb{C}\mathrm{A}$, is the relation defined as:

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

In the same way that we argued for the validity of Eq. 3, we can prove that $\alpha \boxdot \beta$ simulates $\alpha \boxtimes \beta$ for the same identity relation R, if $\langle\!\blacklozenge$ relates α and β:

$$\alpha \langle\!\blacklozenge \beta \textbf{ implies } \alpha \boxtimes \beta \preceq^{R} \alpha \boxdot \beta \tag{4}$$

The following lemma follows from the previous discussion.

Lemma 1 $\alpha \langle\!\blacklozenge \beta$ **implies** $\alpha \boxtimes \beta \approx \alpha \boxdot \beta$

Proof (Outline) We already established $\alpha \boxdot \beta \preceq^{R} \alpha \boxtimes \beta$ in Eq. 3 and, because $\alpha \langle\!\blacklozenge \beta$ is a premise, also $\alpha \boxtimes \beta \preceq^{R} \alpha \boxdot \beta$ holds by Eq. 4. Because R is an identity relation, we have $R = R^{-1}$. We can now apply Definition 3 of $\approx$ to conclude the required result. □

To test whether communication between two CA is actually-cheap, one must pairwise compare their transitions. This is computationally expensive: it requires $\mathcal{O}(n_1 n_2)$ relatively complex checks, where n_1 and n_2 denote the numbers of transitions. This makes the $\langle\!\blacklozenge$-based characterization, in Lemma 1, of when one can safely substitute $\boxtimes$ with $\boxdot$, impractical, although we conjecture this characterization to be complete (essentially by construction). Instead, we now introduce two auxiliary relations on CA, *no-synchronization* and *independence*, which imply actually-cheapness and require fewer (and simpler) checks: only $\mathcal{O}(n_1)$ and $\mathcal{O}(1)$, respectively. Informally, a CA exhibits no-synchronization if it never synchronizes any of its ports (i.e., each of its transitions has a singleton synchronization constraint). For instance, the CA of FIFO in Fig. 4d satisfies no-synchronization. Two CA behave independently if they have disjoint sets of ports. In the next subsection, in an algorithm for computing reasonable partitions, we use these two relations to formulate conditions for deciding which CA should become a member of which part. Although no-synchronization and independence only approximate actually-cheapness, this approximation is precise enough for our purpose: it identifies exactly the synchronous and asynchronous regions of a connector, explained in more detail in Sect. 3.2.

Definition 7 The no-synchronization predicate, denoted by $\xrightarrow{1}$, is the predicate on $\mathbb{C}\mathrm{A}$ defined as:

$$\xrightarrow{1} (Q, \mathcal{P}, \mathcal{M}, \longrightarrow, \iota) \textbf{ iff}$$
$$\big[\big[q \xrightarrow{P, f} q' \textbf{ implies } |P| = 1\big] \textbf{ for all } f, P, q, q'\big]$$

Definition 8 The independence relation, denoted by $\asymp \subseteq \mathbb{C}\text{A} \times \mathbb{C}\text{A}$, is the relation defined as:

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

Intuitively, no-synchronization implies actually-cheapness, because CA with only singleton transitions never cause expensive communication. For instance, suppose $\xrightarrow{1}$ (Bob). Now, if Alice asks Bob if he can fire a transition involving a shared port, Bob can directly answer without derivative communication: either he can fire a transition involving precisely that port, in which case he answers positively, or he cannot. Independence implies actually-cheapness, because CA that do not share any ports do not communicate at all. In principle, we may use an alternative definition for independence in terms of ports involved in transitions instead of our current definition, based on ports in CA. Such a definition of independence would be slightly more powerful (i.e., reveal more automata pairs as independent) in cases where a CA α "knows about a port" but does not actually use it (i.e., there exists a $p \in \mathsf{Port}(\alpha)$, but p does not occur on any of α's transitions). We favor the current version of Definition 8, though, for its simplicity and because such α-s do not occur in practice.

Lemma 2

1. $\xrightarrow{1}(\beta)$ **implies** $\alpha \langle\!\blacklozenge\!\rangle \beta$
2. $\alpha \asymp \beta$ **implies** $\alpha \langle\!\blacklozenge\!\rangle \beta$

Proof (Outline)

1. We split the proof into two parts. In the first part, by expanding our definitions and by applying set theory, we conclude:

$$\left[\begin{bmatrix} q_\beta \xrightarrow{P_\beta, f_\beta} q'_\beta \textbf{ and} \\ P_\beta \cap \mathsf{Port}(\alpha) \neq \emptyset \end{bmatrix} \textbf{ implies } P_\beta \subseteq \mathsf{Port}(\alpha)\right] \textbf{ for all } f_\beta, P_\beta, q_\beta, q'_\beta$$

In words: for all transitions of β, if the ports P_β involved in that transition include a port shared with α, in fact all ports in P_β are shared with α. (This holds, because P_β is a singleton by $\xrightarrow{1}(\beta)$.)
In the second part of the proof, we assume the result of the first part and the premise of the right-hand side in Definition 6 of $\langle\!\blacklozenge\!\rangle$. By applying set theory, we then show for all pairs of transitions involving ports P_α and P_β:

$$\begin{bmatrix} P_\beta \cap \mathsf{Port}(\alpha) = \emptyset \textbf{ and} \\ (\mathsf{Port}(\alpha), P_\alpha) \lozenge (\mathsf{Port}(\beta), P_\beta) \end{bmatrix}$$

$$\textbf{or} \begin{bmatrix} P_\beta \subseteq \mathsf{Port}(\alpha) \textbf{ and} \\ (\mathsf{Port}(\alpha), P_\alpha) \lozenge (\mathsf{Port}(\beta), P_\beta) \end{bmatrix}$$

By expanding the definition of $\lozenge$ in Fig. 6b and by applying set theory, both cases reduce to:

$$(\mathsf{Port}(\alpha), P_\alpha) \blacklozenge (\mathsf{Port}(\beta), P_\beta)$$

Now, because this reduction holds for all pairs of transitions of α and β, we conclude $\alpha \langle\!\blacklozenge\!\rangle \beta$.

2. In addition to $\alpha \asymp \beta$, we assume the premise of the right-hand side in Definition 6 of $\langle\!\blacklozenge\!\rangle$. Then, by expanding Definition 8 of $\asymp$ and the definition of $\lozenge$ in Fig. 6b, we conclude for all pairs of transitions involving ports P_α and P_β:

$$\begin{array}{l} \mathsf{Port}(\alpha) \cap \mathsf{Port}(\beta) = \emptyset \textbf{ and} \\ \mathsf{Port}(\alpha) \cap P_\beta = \mathsf{Port}(\beta) \cap P_\alpha \end{array}$$

Because $P_\alpha \subseteq \mathsf{Port}(\alpha)$ and $P_\beta \subseteq \mathsf{Port}(\beta)$ (implicitly assuming that α and β are well-defined CA), we conclude $\mathsf{Port}(\alpha) \cap P_\beta = \emptyset = \mathsf{Port}(\beta) \cap P_\alpha$. Finally, by applying the definition of $\blacklozenge$ in Fig. 6b and by observing that the previous reduction holds for all pairs of transitions of α and β, we conclude $\alpha \langle\!\blacklozenge\!\rangle \beta$. □

The combination of Lemmas 1 and 2 establishes that under no-synchronization or independence, substituting $\boxtimes$ with $\boxdot$ in a product term consisting of *exactly* two CA is sound. Our next theorem generalizes this to an arbitrary number of CA. Notationally, to avoid excessive parentheses in the following theorem and its proof, we assume right associativity for $\boxdot$. For instance, we write $\alpha \boxdot \beta \boxdot \gamma \boxdot \delta$ instead of $\alpha \boxdot (\beta \boxdot (\gamma \boxdot \delta))$. This notational convention is important, because $\boxdot$ is *not* algebraically associative (in contrast to $\boxtimes$). See the end of this subsection for a further remark on this matter.

Theorem 3 (Substitution is sound)

$$\left[\begin{bmatrix} \xrightarrow{1}\beta_i \textbf{ for all} \\ 1 \leq i \leq m \end{bmatrix} \textbf{ and } \begin{bmatrix} [i \neq j \textbf{ implies } \gamma_i \asymp \gamma_j] \\ \textbf{for all } 1 \leq i, j \leq n \end{bmatrix}\right]$$

$$\textbf{implies } \begin{array}{c} \beta_1 \boxtimes \cdots \boxtimes \beta_m \boxtimes \gamma_1 \boxtimes \cdots \boxtimes \gamma_n \\ \approx \beta_1 \boxdot \cdots \boxdot \beta_m \boxdot \gamma_1 \boxdot \cdots \boxdot \gamma_n \end{array}$$

Proof (Outline) We prove the equation from right (i.e., γ_n) to left (i.e., β_1). First, because $\gamma_{n-1} \asymp \gamma_n$ by the premise and by applying Lemmas 1 and 2:1, we conclude $\gamma_{n-1} \boxtimes \gamma_n \approx \gamma_{n-1} \boxdot \gamma_n$. Now, to similarly prove $\gamma_{n-2} \boxtimes \gamma_{n-1} \boxtimes \gamma_n \approx \gamma_{n-2} \boxdot \gamma_{n-1} \boxdot \gamma_n$ (by using Lemmas 1 and 2:1), we must show $\gamma_{n-2} \asymp \gamma_{n-1} \boxdot \gamma_n$. By expanding Definition 8 of $\asymp$ and by applying set theory, we can conclude this from $[\gamma_{n-2} \asymp \gamma_{n-1} \textbf{ and } \gamma_{n-2} \asymp \gamma_n]$ in the premise. Using a straightforward inductive argument, we therefore can show $\gamma_1 \boxtimes \cdots \boxtimes \gamma_n \approx \gamma_1 \boxdot \cdots \boxdot \gamma_n$. A similar inductive argument, in which we use Lemmas 1 and 2:2, concludes the proof (Lemma 2:2 is applicable regardless of the relation between the CA involved). □

Through Theorem 3, we thus showed that if a partitioning algorithm constructs a partition $\mathcal{A}$ such that $\mathcal{A}$ contains m parts $B_1, \ldots, B_m$ with property $\xrightarrow{1} [\![B_i]\!]$ and n parts $C_1, \ldots, C_n$ with property $[\![C_i]\!] \asymp [\![C_j]\!]$, only cheap communication (modeled by $\boxdot$) instead of expensive communication (modeled by $\boxtimes$) is necessary among their corresponding CA-implementations at run time, while collectively, they are behaviorally equivalent to $[\![X]\!]$. In other words, such an algorithm computes reasonable partitions according to our cheap/expensive guideline. Formally, the following equations hold:

$$\begin{aligned}
[\![X]\!] &= [\![\mathcal{A}]\!] \\
&= [\![\{B_1, \ldots, B_m, C_1, \ldots, C_n\}]\!] \\
&= [\![\{[\![B_1]\!], \ldots, [\![B_m]\!], [\![C_1]\!], \ldots, [\![C_n]\!]\}]\!] \\
&= \bigboxtimes \{[\![B_1]\!], \ldots, [\![B_m]\!], [\![C_1]\!], \ldots, [\![C_n]\!]\} \\
&= [\![B_1]\!] \boxtimes \cdots \boxtimes [\![B_m]\!] \boxtimes [\![C_1]\!] \boxtimes \cdots \boxtimes [\![C_n]\!] \\
&\approx [\![B_1]\!] \boxdot \cdots \boxdot [\![B_m]\!] \boxdot [\![C_1]\!] \boxdot \cdots \boxdot [\![C_n]\!]
\end{aligned}$$

Earlier, we mentioned that $\boxdot$ does not exhibit algebraic associativity. This means that generally, the particular order in which CA-implementations communicate with each other at run-time matters: communication must start between those CA most deeply nested in the previous equations (i.e., with our right associative notation, the rightmost two CA $[\![C_{n-1}]\!]$ and $[\![C_n]\!]$). This limitation can degrade run-time performance and seems a serious practical problem. To solve this, we can extend our theory to compensate for nonassociativity of $\boxdot$ along the same lines as [29, Section 5]. There, Jongmans and Arbab describe a solution to this problem for CA without data constraints, which straightforwardly generalizes to our current setting with unrestricted CA. The idea is to study and prove under which circumstances rearranging CA is sound such that different CA can start communication at different times.

To roughly explain this, first consider the case of $\boxtimes$, which does exhibit associativity. Suppose that we want to implement $\alpha_1 \boxtimes (\alpha_2 \boxtimes \alpha_3)$ as three parallel processes—one for every α_i—that need to communicate to synchronize their transitions. If we want to be very precise, then in this implementation, α_2 and α_3 must communicate with each other before they can communicate with α_1. (This corresponds to the automata-theoretic idea that one would first compute the nested product of α_2 and α_3 to evaluate the whole expression.) However, because $\boxtimes$ is associative, any implementation that first lets α_1 and α_2 communicate is, in fact, equally fine: structurally, these implementations differ, but behaviorally, they coincide. In other words, as long as the implementation behaves as $\alpha_1 \boxtimes (\alpha_2 \boxtimes \alpha_3)$ modulo associativity and commutativity, there is no problem. Formally, if $\approx_{\text{AC}}$ denotes behavioral equivalence up to associativity and commutativity, we can take the quotient $\mathbb{C}\text{A}/\approx_{\text{AC}}$ and require that every implementation of $\alpha_1 \boxtimes (\alpha_2 \boxtimes \alpha_3)$ behaves as *some* element in its equivalence class. With a bit more technical machinery, we can subsequently prove that it is fine for an implementation to "switch" between elements in an equivalence class during execution, which models the fact that at run time, communication does not always have to start between the same automata (e.g., sometimes we first have communication between α_1 and α_2 and other times between α_2 and α_3) to get equivalent behavior.

Now, let us consider the case of $\boxdot$. Even though $\boxdot$ is not generally associative, we can show that under certain conditions, associativity does apply. For instance, $\alpha_1 \boxdot (\alpha_2 \boxdot \alpha_3) \approx (\alpha_1 \boxdot \alpha_2) \boxdot \alpha_3$ whenever both α_1 and α_2 are independent of α_3 and α_1 satisfies no-synchronization (but the conditions are more general than this). More formally, if $\approx_{\text{cAC}}$ denotes behavioral equivalence up to this kind of *conditional associativity* and commutativity, we can take the quotient $\mathbb{C}\text{A}/\approx_{\text{cAC}}$ in the same way as with $\boxtimes$ above. Practically, conditional associativity turns out to cover all cases that we care about: the previous formal model can be used to show that the particular order in which CA-implementations communicate with each other at run time does not matter for those CA-implementations that we actually encounter in practice. Full technical details appear elsewhere [30].

Finally, we remark that all lemmas and theorems in this subsection are generalizations, to unrestricted CA, of results in the local product theory for CA without data constraints [29]. In fact, those previous results induce a characterization (of when one can safely substitute $\boxtimes$ with $\boxdot$) strictly more general than the one in Theorem 3 (modulo data) but with a less intuitive interpretation. From our current perspective and for our current audience, however, extending only a subset of that characterization with data seemed more practically relevant and natural to explain. Working out these generalizations was laborious but relatively straightforward. More interestingly, however, we now recognize that the previous definitions, lemmas, and theorems in fact exemplify a novel proof method for establishing when substituting one distributed algorithm (e.g., an expensive one) with another distributed algorithm (e.g., a cheap one) is semantics-preserving, by analyzing properties of product operators on automata that model those algorithms.

3.2 Hybrid-code generator

The new task of a hybrid-code generator, over and beyond the tasks of a centralized-code generator [31], is dividing the CA of the k primitive constituents of the input connector over parts in a partition $\mathcal{A}$; it can subsequently use existing techniques to generate code for every part in $\mathcal{A}$. To carry out this new task, a hybrid-code generator can run the algorithm in Fig. 7 with time complexity upper bound of $\mathcal{O}(k^2)$.

Our algorithm iterates over an indexed input set of CA. In each iteration, either it puts the current CA α_i in a new part in $\mathcal{B}_i$ (if α_i satisfies no-synchronization), or it computes

```
function PARTITION({α_1 , ... , α_k})
 (B_0 , C_0) := (∅ , ∅)
 for all 1 ≤ i ≤ k do
  if -1->α_i then
   (B_i , C_i) := (B_{i-1} ∪ {{α_i}} , C_{i-1})
  else
   C̄_i := {C ∈ C_{i-1} | ∃γ ∈ C and α_i ≁ γ}
   (B_i , C_i) := (B_{i-1} , (C_{i-1} \ C̄_i) ∪ {{α_i} ∪ ⋃C̄_i})
 return (B_k , C_k)
```

Fig. 7 Algorithm for computing reasonable partitions

a new part for α_i in $\mathcal{C}_i$, possibly including existing parts, such that the new part contains all CA dependent on α_i. The following theorem establishes the algorithm's correctness: it yields a partition of the input set of CA *and* the partition is in fact reasonable according to our cheap/expensive guideline (i.e., the interpretation of the parts in the partition satisfy the premise of Theorem 3). Note that the algorithm terminates because the loop is bounded by k.

Theorem 4 (Correctness of Figure 7)

$\textsc{Partition}(X) = (\mathcal{B}_k, \mathcal{C}_k)$ **implies**

1. $\mathcal{B}_k \cup \mathcal{C}_k$ *is a partition of* X
2. $\xrightarrow{1} [\![B]\!]$ **for all** $B \in \mathcal{B}_k$
3. $\big[C \neq C'$ **implies** $[\![C]\!] \asymp [\![C']\!]\big]$ **for all** $C, C' \in \mathcal{C}_k$

Proof (outline)

1. We have the following proof obligation:

$$\bigcup \mathcal{B}_k \cup \bigcup \mathcal{C}_k = X \text{ and}$$
$$\Big[\Big[A \neq A' \text{ and } A, A' \in \mathcal{B}_k \cup \mathcal{C}_k \text{ implies } A \cap A' = \emptyset\Big] \text{ for all } A, A'\Big]$$

   By induction on $0 \leq i \leq k$, we prove the following stronger property:

$$\mathcal{B}_{i-1} \cap \mathcal{C}_{i-1} = \emptyset \text{ and}$$
$$(\bigcup \mathcal{B}_{i-1} \cup \bigcup \mathcal{C}_{i-1}) \cap \{\alpha_i, \ldots, \alpha_k\} = \emptyset \text{ and}$$
$$\bigcup \mathcal{B}_{i-1} \cup \bigcup \mathcal{C}_{i-1} = \{\alpha_1, \ldots, \alpha_{i-1}\} \text{ and}$$
$$\Big[\Big[A \neq A' \text{ and } A, A' \in \mathcal{B}_i \cup \mathcal{C}_i \text{ implies } A \cap A' = \emptyset\Big] \text{ for all } A, A'\Big]$$

   The base case (i.e., $\mathcal{B}_0 = \mathcal{C}_0 = \emptyset$) follows straightforwardly. To prove the inductive step, in which the algorithm considers α_i, we distinguish two cases. The easy one is $\xrightarrow{1} (\alpha_i)$. In that case, the algorithm adds $\{\alpha_i\}$ to $\mathcal{B}_i$ while leaving $\mathcal{C}_i$ unchanged. By using set theory and the induction hypothesis for $i-1$, we can straightforwardly establish the induction hypothesis for i. The more difficult case is $\big[\textbf{not } \xrightarrow{1} (\alpha_i)\big]$, because we must show that no sets in $\mathcal{C}_{i-1} \setminus \overline{\mathcal{C}}_i$ and $\{\{\alpha_i\} \cup \bigcup \overline{\mathcal{C}}_i\}$ overlap. We do so in a proof by contradiction. In particular, if two overlapping sets exist, those sets must have a shared CA α, and either $\alpha = \alpha_i$ or $\alpha \neq \alpha_i$. Both cases reduce to **false** by using set theory and the induction hypothesis for $i-1$, from which we conclude that overlapping sets cannot exist. Then, we can establish the induction hypothesis for i also for $\big[\textbf{not } \xrightarrow{1} (\alpha_i)\big]$. Finally, the induction hypothesis for k implies the required result, which concludes the proof.
2. The proof is nearly trivial: one can derive the required result almost directly from the definition of the algorithm. To formally prove it, we proceed by induction on $1 \leq i \leq k$. In the inductive step, we establish that the algorithm adds α_i to $\mathcal{B}_i$ only if $\xrightarrow{1} (\alpha_i)$ and leaves $\mathcal{B}_i$ unchanged otherwise.
3. We prove the required result by induction on $1 \leq i \leq k$. The base case holds vacuously (quantification over the empty set $\mathcal{C}_0$). To prove the inductive step, in which the algorithm considers α_i, we distinguish two cases. If $\xrightarrow{1} (\alpha_i)$, the algorithm leaves $\mathcal{C}_i$ unchanged, and the induction hypothesis for $i-1$ directly applies to i. Otherwise, if $\big[\textbf{not } \xrightarrow{1} (\alpha_i)\big]$, we tentatively assume that $C_1, C_2 \in \mathcal{C}_i$ exist such that $[\![C_1]\!] \not\asymp [\![C_2]\!]$. By the definition of the algorithm, we conclude $C_1, C_2 \in (\mathcal{C}_{i-1} \setminus \overline{\mathcal{C}}_i) \cup \{\{\alpha_i\} \cup \bigcup \overline{\mathcal{C}}_i\}$. Now, if C_1 and C_2 both come from $\mathcal{C}_{i-1} \setminus \overline{\mathcal{C}}_i$, they both come from $\mathcal{C}_{i-1}$, and consequently, we can immediately conclude **false** by the induction hypothesis for $i-1$. Thus, either C_1 or C_2 must be equal to $\{\alpha_i\} \cup \bigcup \overline{\mathcal{C}}_i$ (the case that C_1 and C_2 are equal contradicts the premise of this theorem). Suppose it is C_1. Then, from our tentative assumption $[\![C_1]\!] \not\asymp [\![C_2]\!]$, we can conclude that an $\alpha \in C_1$ exists such that $\alpha \not\asymp [\![C_2]\!]$. If $\alpha = \alpha_i$, we conclude **false**, because α_i was considered by the algorithm only in the ith iteration and therefore cannot be a member of C_1 yet (which was constructed in an iteration before the ith). Otherwise, if $\alpha \in \bigcup \overline{\mathcal{C}}_i$, a $C_1' \in \overline{\mathcal{C}}_i \subseteq \mathcal{C}_{i-1}$ exists such that $\alpha \in C_1'$. Moreover, because $\alpha \not\asymp [\![C_2]\!]$ and $\alpha \in C_1$, we can show $[\![C_1']\!] \not\asymp [\![C_2]\!]$. This, however, contradicts the induction hypothesis for $i-1$, and we conclude **false**. Thus, our tentative assumption was wrong, and we conclude $[\![C_1]\!] \asymp [\![C_2]\!]$. Finally, the induction hypothesis for k implies the required result, which concludes the proof. □

Interestingly, the reasonable partitions $\mathcal{A} = \mathcal{B} \cup \mathcal{C}$ computed by the algorithm in Fig. 7 correspond to the *synchronous* and the *asynchronous regions* of a connector [59,60]. First, every part $B \in \mathcal{B}$ represents an asynchronous region (e.g., a FIFO primitive): the fact that $[\![B]\!]$ has only singleton synchronization constraints (i.e., $\xrightarrow{1} [\![B]\!]$) models that its ports can*not*—neither intentionally nor coincidentally—

Table 2 Evolution of $\mathcal{B}_i$ and $\mathcal{C}_i$ during the execution of the algorithm in Fig. 7 on the set of CA in Fig. 8

i	$\mathcal{B}$	$\mathcal{C}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\{\{\alpha_1\}\}$
2	$\emptyset$	$\{\{\alpha_1, \alpha_2\}\}$
3	$\emptyset$	$\{\{\alpha_1, \alpha_2, \alpha_3\}\}$
4	$\{\{\alpha_4\}\}$	$\{\{\alpha_1, \alpha_2, \alpha_3\}\}$
5	$\{\{\alpha_4\}\}$	$\{\{\alpha_1, \alpha_2, \alpha_3\}, \{\alpha_5\}\}$
6	$\{\{\alpha_4\}\}$	$\{\{\alpha_1, \alpha_2, \alpha_3\}, \{\alpha_5\}, \{\alpha_6\}\}$
7	$\{\{\alpha_4\}\}$	$\{\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_7\}, \{\alpha_5\}\}$
8	$\{\{\alpha_4\}, \{\alpha_8\}\}$	$\{\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_7\}, \{\alpha_5\}\}$
9	$\{\{\alpha_4\}, \{\alpha_8\}\}$	$\{\{\alpha_1, \alpha_2, \alpha_3, \alpha_6, \alpha_7\}, \{\alpha_5\}, \{\alpha_9\}\}$

synchronize at run time. Dually, every part $C \in \mathcal{C}$ represents a synchronous region (e.g., the sequence of three Sync primitives modeled by Alice, Bob, and Carol in a previous example): by the duality of (a)synchronous regions, a CA for a synchronous region has at least one transition with a nonsingleton synchronization constraint. When such a transition fires, synchronization between at least two ports takes place.

As an example of the execution of the algorithm, we consider ExampleConn in Fig. 1e. Figure 8 shows nine CA, one for every channel ($\alpha_2, \alpha_4, \alpha_7$, and α_8) and every node (α_1, α_3, α_5, α_6, and α_9) that ExampleConn consists of. We model boundary nodes A, B, C, and D with pairs of an input port and an output port (essentially synchronous channels), one of which the environment uses for I/O; we model internal node X implicitly as a composition of Merger and Replicator primitives as explained in Sect. 2. Let the CA in Fig. 8 constitute the input set. Table 2 shows the evolution of sets $\mathcal{B}$ and $\mathcal{C}$ as the algorithm progresses (for simplicity, we omit subscripts from $\mathcal{B}$ and $\mathcal{C}$).

Initially, both sets are empty. In the first iteration, because α_1 violates no-synchronization, the algorithm adds $\{\alpha_1\}$ to $\mathcal{C}$. In the second iteration, first, the algorithm concludes that α_2 violates no-synchronization. Subsequently, because α_1 and α_2 share port A_{out}, they are dependent (i.e., $\alpha_1 \not\asymp \alpha_2$), and therefore, the algorithm includes $\{\alpha_1\}$ in $\overline{\mathcal{C}}_2$. Finally, it removes $\overline{\mathcal{C}}_2$ from $\mathcal{C}$ and adds $\{\{\alpha_2\} \cup \{\alpha_1\}\}$ to $\mathcal{C}$. In the third iteration, something similar happens: α_2 and α_3 share port X_{in1}. Therefore, the algorithm includes $\{\alpha_1, \alpha_2\}$ in $\overline{\mathcal{C}}_3$ (note that it does not matter that α_1 and α_3 have no shared ports). In the fourth iteration, the algorithm concludes that α_4 satisfies no-synchronization: α_4 models one of the two FIFOs in the connector, which, as we explained before, by itself constitutes an asynchronous region. The algorithm subsequently adds $\{\alpha_4\}$ to $\mathcal{B}$ (and leaves $\mathcal{C}$ unchanged). In the fifth iteration, the algorithm adds $\{\alpha_5\}$ to $\mathcal{C}$, because α_5 violates no-synchronization and shares no ports with α_1, α_2, or α_3. Similarly, the algorithm adds $\{\alpha_6\}$ to $\mathcal{C}$ in the sixth iteration. In the seventh iteration, the algorithm first concludes that α_7 violates no-synchronization. Now, α_7 shares a port with α_6 (i.e., B_{out}) and with α_3 (i.e., X_{in2}). The algorithm therefore includes both $\{\alpha_1, \alpha_2, \alpha_3\}$ and $\{\alpha_6\}$ in $\overline{\mathcal{C}}_7$, and after removing the resulting $\overline{\mathcal{C}}_7$ from $\mathcal{C}$, it adds $\{\{\alpha_7\} \cup \{\alpha_1, \alpha_2, \alpha_3\} \cup \{\alpha_6\}\}$ to $\mathcal{C}$. The remaining iterations of the algorithm proceed similarly.

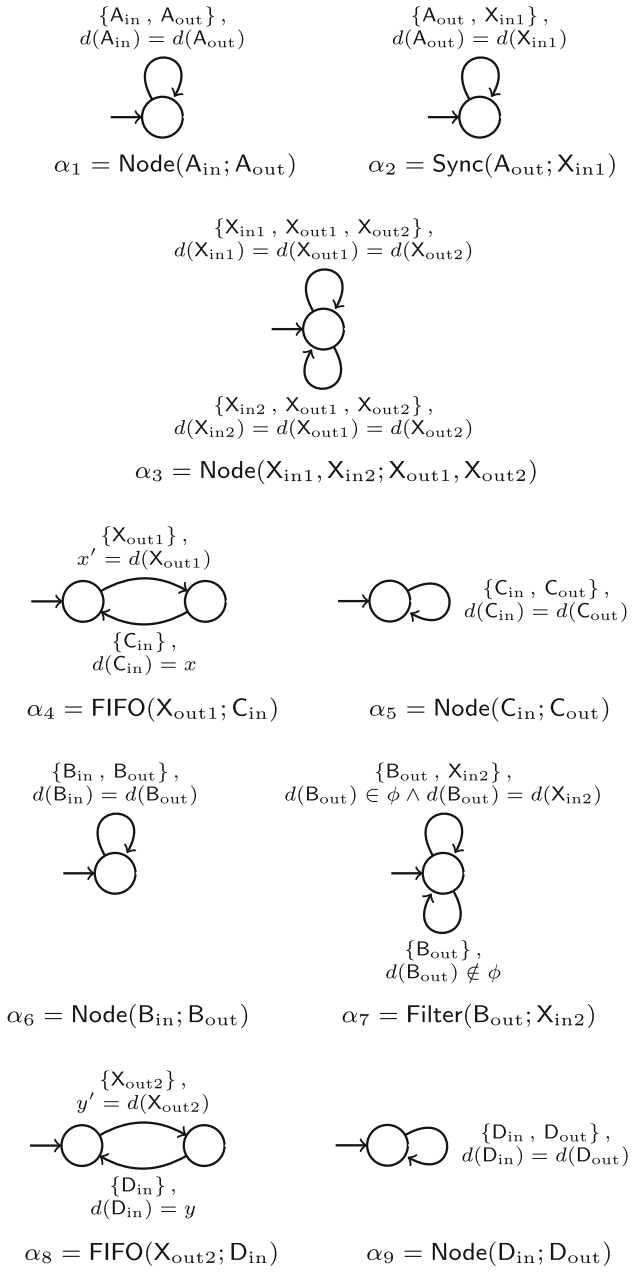


Fig. 8 Constraint automata for the channels and nodes in ExampleConn in Fig. 1e

After the algorithm terminates, the code generator computes the interpretation of the parts in the constructed partition. This yields one new CA, namely for part $\{\alpha_1, \alpha_2, \alpha_3, \alpha_6$ $\alpha_7\}$ (the interpretation of the four singleton parts, two in $\mathcal{B}$ and two in $\mathcal{C}$, each is itself). After generating code, we thus obtain five CA-implementations, executed in parallel at run

time. Now, one of the advantages of this hybrid approach is that the implementations of α_5 and α_9 can simultaneously each make a transition. In particular, *while* the implementation of α_5 makes its transition (which drains the buffer from the upper FIFO in Fig. 1e), the implementation of α_9 can start its transition (which drains the buffer from the lower FIFO); it does not need to wait until the α_5-implementation finishes. In the purely centralized approach, this cannot happen: as soon as a single, sequential, big CA-implementation starts a transition, other enabled transitions must wait, until the started transition ends.

The previous example shows that partitions computed by our algorithm can be *imbalanced*: in the example, the computed partition has one big part (consisting of five CA) and four small parts (consisting of only one CA). One may wonder about whether this is problematic. As stated in the introduction, the purpose of the hybrid approach (i.e., partitioning) is twofold: it should strike a middle ground between latency and parallelism at run time while achieving reasonably fast compilation at build time. The cheap/expensive communication guideline that we adopted for partitioning covers the first property by its very definition, regardless of whether the resulting partitions are balanced or not. In contrast, the second property may suffer from imbalanced partitions: if one subset in a partition contains too many "inconveniently shaped" automata, the product of those automata may become too large to compute or to store. For instance, if we were to put n independent FIFOs in the same subset in a partition, the state space of their product contains 2^n states. Quickly as n increases, then compilation takes not only too much time but also more memory than is available.

The crucial question is therefore as follows: Do the imbalanced partitions that our algorithm computes contain subsets with too many "inconveniently shaped" automata? As far as state space explosion is concerned, the answer is no. The only primitive whose automaton has more than one state is FIFO, and using our partitioning algorithm, every FIFO constitutes its own subset. On the other hand, subsets with more than one automaton are guaranteed to consist of only single-state automata. Because the product of any number of single-state automata yields a single-state automaton, state space explosion will never occur. Even if we later decide to add a new primitive whose automaton has more than one state and which does not satisfy no-synchronization (such that it will not constitute its own subset), Baier et al. have shown that we can always break the automaton of this new primitive down into a number of single-state automata and a number of FIFOs [3], as long as we assume a finite data domain.

Thus, imbalanced partitions will not be problematic for state space explosion. However, another form of subset with too many "inconveniently shaped" automata exists, which causes the transition relation to explode. This problem, presented in more detail in Sect. 6.4, can be solved in some cases, but generally, it seems to require resorting to the purely distributed approach at the cost of run time overhead. The fact that in some cases imbalanced partitions are problematic, however, does not make our splitting algorithm useless: it at least identifies the trouble spots in the system, which may require special treatment. This means that it is unfortunately not always possible to apply only our splitting algorithm (at least not in its current form) to obtain a hybrid implementation that satisfies both desired properties. We are trying to find better solutions, but we consider this as a different research question than the subject of this paper.

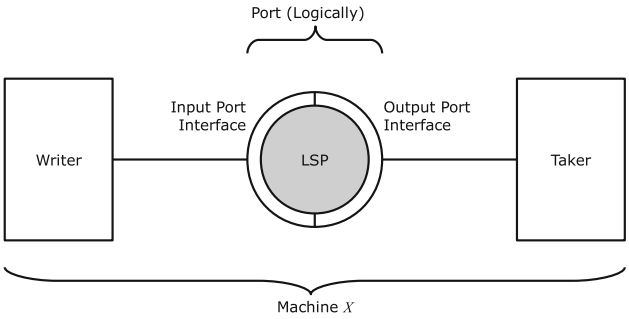
4 Implementation: practical realization

4.1 Hybrid connectors

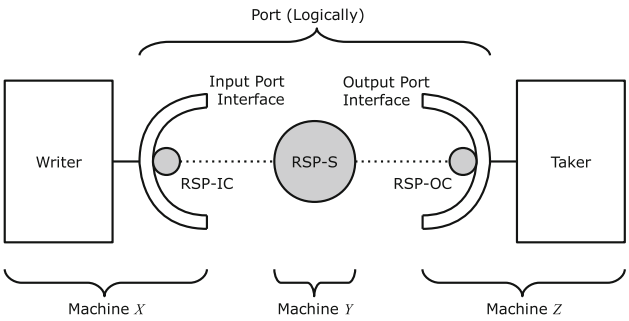
We extended an existing Reo-to-Java code generation tool [31], which translates CA to implementations of Java's `Runnable` interface. Every such a CA-implementation can run in its own Java thread. The control-flow inside the main `run()` method follows a conceptually simple (event-driven) state machine pattern. Details appear elsewhere [31]; here, we focus on the process of firing a transition $q \xrightarrow{P,f} q'$. The following enumeration summarizes this process.

1. Check synchronization constraint P.
   (a) Check the ports in P for pending I/O. Let $P' \subseteq P$ denote the set of ports without pending I/O.
   (b) Ask neighbors with ports in P' which data constraints must hold for them to be ready for I/O on those ports. Let F denote that set of data constraints.
2. Solve extended data constraint $f \wedge \bigwedge F$.
3. Commit and conclude.
   (a) Distribute data items among pending take operations according to the solution found for $f \wedge \bigwedge F$.
   (b) Mark pending I/O completed and update state to q'.
   (c) Perform I/O on ports in P' according to the solution found for $f \wedge \bigwedge F$.
   (d) Notify neighbors with ports in P' that they must fire their transitions involving those ports.
   (e) Await the completion of those transition.

During steps (1) and (2), a CA-implementation *can* still abort the firing process, and it *will* do so if the synchronization constraint does not hold—in which case F contains $\bot$—or if the data constraint has no solution under the then-pending I/O operations. Once step (3) starts, however, the transition *must* run to completion. To ensure that the whole firing process runs atomically, that pending I/O operations do not timeout during the firing process, and that neighbors do not change state, a CA-implementation uses a two-phase locking scheme [9].



(a) Local synchronization point (shared-memory port)



(b) Remote synchronization point (distrib.-memory port)

Fig. 9 Synchronization points

Ports are implemented as interfaces that provide access to concurrent data structures called *synchronization points* [31]. Essentially, synchronization points register pending write and take operations. There exist two kinds of port interfaces: input ports expose only `write(...)` methods for performing write operations, while output ports expose only `take(...)` methods for performing take operations. Application developers can use input and output ports for letting concurrent fragments of their computation code interact with each other via a CA-implementation (i.e., via a connector). Internal classes in the Reo run-time libraries, as well as generated CA-implementations, call also other methods on input and output ports (e.g., checking for pending I/O operations, communicating with neighbors via synchronization points).

The run-time libraries of the original implementation of the Reo-to-Java code generator contain only one *shared-memory* implementation of input and output ports. Figure 9a shows an infographic, where "LSP" stands for "local synchronization point." This name reflects that the Java objects constituting such a synchronization point live on the same machine as the CA-implementations that access that synchronization point through input and output ports. Logically, a triple of an LSP, an input port, and an output port makes up a single port.

To enable developers to fully exploit the improved parallelism in the hybrid connector implementations generated by our tool extension (cf. the centralized approach), we implemented a new *distributed-memory* implementation of input and output ports. In particular, this allows developers to deploy connector implementations generated by our tool extension on different machines in a network. Figure 9b shows an infographic, where "RSP" stands for "remote synchronization point," "- S" for "server," "- IC" for "input client," and "- OC" for "output client." Deployment of a remote synchronization point starts with deploying an RSP- S on some machine in the network. Essentially, an RSP- S is a Web service, implemented using JAX- WS,[1] whose operations provide its clients access to a "classical" LSP inside of it. Once an RSP- S has been deployed, one can construct input and output ports to access it, including input and output clients. During execution, those port interfaces use such clients for delegating, to the deployed RSP- S, those method calls that they cannot process locally (e.g., checking for pending I/O operations). The CA-implementations that call methods on port interfaces do not know whether those calls require network communication: whether synchronization points accessed through port interfaces run locally or remotely is completely transparent.

4.2 Hybrid-code generator

Our extension to the Reo-to-Java code generator is—as the original—implemented in Java as an Eclipse plug-in. This plug-in depends on the *Extensible Coordination Tools* (ECT): a collection of Eclipse plug-ins that constitute an IDE for Reo.[2] Our extension outputs ℓ Java classes—one for every part in the reasonable partition computed using the algorithm in Fig. 7—each of which implements the `Runnable` interface. Instances of those classes can be deployed on different machines and connected to each other via distributed-memory ports (including the required remote synchronization point servers and clients). For testing purposes, the code generator also generates a default main program which deploys an instance of each of the ℓ classes and each of the required distributed-memory ports on the same machine. Indeed, our implementation of distributed-memory ports works also for machines X, Y, and Z in Fig. 9b such that $X = Y = Z$. However, in that case, using shared-memory ports is more sensible.

For performance reasons, we simplified the implementation of the algorithm in Fig. 7 and the actual code generation process (using ANTLR's StringTemplate engine [54]) by exploiting the observation that the only primitive currently supported by the ECT that satisfies the first condition in Theorem 3 is FIFO. This, for instance, reduced checking the condition of the **if**-statement in Fig. 7 from iterating over all transitions of a CA to checking that CA's type.

[1] http://jax-ws.java.net.

[2] http://reo.project.cwi.nl.

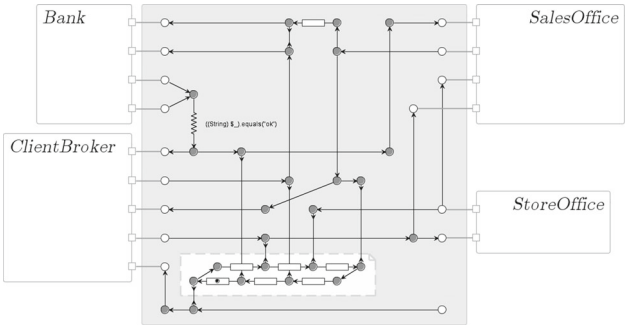


Fig. 10 Connector for orchestrating the four WSs in the online purchase scenario

5 Case study

We proceed with a case study in the domain of distributed service orchestration. The purpose of this section is to discuss, present, and exemplify opportunities that the theory we previously presented provides in practice. (We do not try to reexplain or quantitatively evaluate our theory.)

In the following case study, we elaborate on the same example as given in [37], which implements a classical online purchase scenario. This interaction involves four Web services (WS) named *ClientBroker*, *StoreOffice*, *SalesOffice*, and *Bank*. The *ClientBroker* service takes care of interfacing a client to the other services, which deal with: the information about the store (i.e., the *StoreOffice* service), the procedure to prepare the invoice (i.e., the *SalesOffice* service), and the effective payment management (i.e., the *Bank* service). Using the orchestration approach to Web service composition [57], we compose these four services so as to realize the composite behavior described below.

First, an end user provides the *ClientBroker* service a description of a product that he or she wants to purchase. The *ClientBroker* subsequently searches for a product that matches the description and returns the result. This result should be routed to both the *StoreOffice* and the *SalesOffice* (by the orchestrator), effectively placing an order for that product. Upon receiving the order, the *StoreOffice* checks if the ordered product is still in stock. The output of the *StoreOffice* ("yes"/"no" plus additional price information) should be routed to the *SalesOffice*. Once the *SalesOffice* has received both the order and the stock information, it computes and outputs the final price. This final price should be routed to both the *ClientBroker* (to confirm that it does not exceed the range specified by the end user) and the *Bank*. The *ClientBroker* now outputs the credit card number of the end user, which should also be routed to the *Bank*. With the price and the credit card number, the *Bank* finally completes the transaction by processing the payment. See [37] for a more detailed description.

Figure 10 shows a Reo connector, named Orchestrator, for orchestrating the four WSs according to the previous operational description of the scenario. In this case, a Reo expert designed this connector by hand. Alternatively, depending on the expertise available in an organization, one can specify the orchestration protocol as an automaton, as a BPEL program, or as a UML sequence/activity diagram and use mechanical connector synthesis technology to obtain this (or a behaviorally equivalent) connector [3,17].

Jongmans et al. [37] generate a centralized implementation of Orchestrator and deploy that implementation on a single machine in a network. We significantly improve on that by using our hybrid-code generator to obtain a hybrid implementation and deploy it across multiple machines.

First, our code generator establishes that Orchestrator consists of 43 channels and 42 nodes. It then matches each of those constituents with a CA describing that constituent's behavior, which yields $k = 85$ small CA. For instance, Fig. 11 shows seven CA describing the constituents in the largest colored region in Fig. 13. Next, our code generator runs the algorithm in Fig. 7 to obtain a reasonable partition. This partition consists of thirteen parts: seven singleton parts for asynchronous regions (i.e., FIFOs, which satisfy $\xrightarrow{1}$) and six parts for synchronous regions. For instance, the CA in Fig. 11 form a complete part of the latter kind (none of those seven CA shares a port with any CA in any other part). The code generator then computes, for every part, the $\boxtimes$-product of the CA

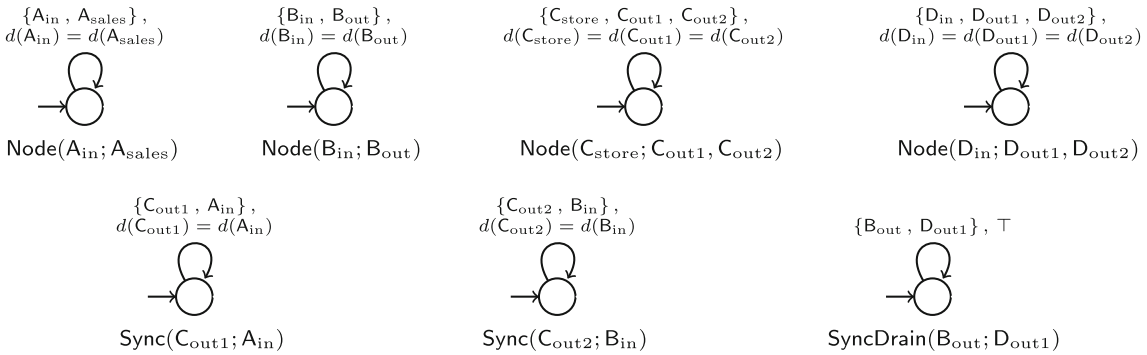
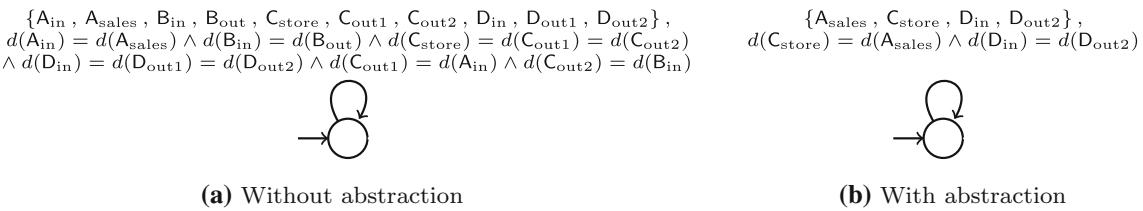


Fig. 11 Constraint automata for the channels and nodes in the largest colored region in Fig. 13; nodes are named A, B, C, and D, from top to bottom and from left to right (in these CA semantics, every node is represented by a number of input and output ports, which are shared with channels)



(a) Without abstraction **(b)** With abstraction

Fig. 12 ⊠-product of the CA in Figure 11, with and without abstracting away internal ports

in that part. This yields $\ell = 13$ medium CA. For instance, Fig. 12 shows the ⊠-product of the CA in Fig. 11. Finally, the code generator compiles the thirteen medium CA to as many CA-implementations. For instance, at run time, the generated implementation of the CA in Fig. 12 has only one execution step (i.e., transition) that it repeats infinitely often: It atomically transports a piece of data from *StoreOffice* to *SalesOffice*, and simultaneously, it transports a piece of data from the output port of one FIFO to the input port of another one. The latter ensures that the generated CA-implementation can perform this execution step only if the whole connector is in a state that allows this (i.e., if the FIFOs are, respectively, full and empty). See [37] for details.

Next, we discuss the deployment of the generated code. Logically, we have five machines: called Red (for the actual client), Green (for *ClientBroker*), Blue (for *StoreOffice*), Cyan (for *SalesOffice*), and Magenta (for *Bank*). Although any distribution of the thirteen generated CA-implementations (and the ports between them, through which CA-implementations communicate with each other) over the five available machines would technically work, some of those distributions make more sense than others. Generally, the problem of (automatically) optimally distributing CA-implementations over machines is an interesting research challenge, which we regard as important future work (see also Sect. 9). For now, we adopted the following ad hoc approach: to minimize network traffic, we manually distributed CA-implementations in such a way that every piece of data goes over the network exactly once. In contrast, in the centralized implementation in [37], every piece of data goes over the network twice: first from the sending WS machine to the Orchestrator machine and then from the Orchestrator machine to the receiving WS machine. Thus, the hybrid approach can improve the centralized approach not only in terms of run-time parallelism but, as a consequence of less network traffic and no single-point contention, also in terms of latency (especially with large data). Figure 13 shows the deployment of parts of the orchestration on machine Blue, and Fig. 14 shows the full deployment on all machines.

For deploying ports, we first analyzed which ports are shared between CA-implementations running on the same machine. We deployed those ports as shared-memory ports (similar to [37]) and all other ports as distributed-memory ports. For every distributed-memory port p, we deployed

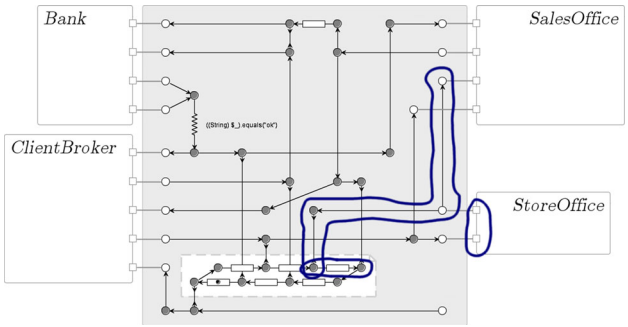


Fig. 13 Deployment of medium CA-implementations and the *StoreOffice* service on machine Blue

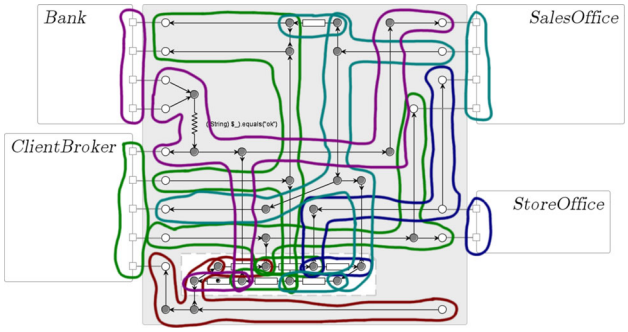


Fig. 14 Full deployment on all machines

its remote synchronization point server (RSP-S) on the same machine as the CA-implementation that uses p's input interface. This means that pieces of data involved in write operations on p stay on that machine as long as no other CA-implementation (running on a different machine) wants to take that data from p's RSP-S (via p's output interface). Consequently, pieces of data travel over the network only if absolutely necessary.

To actually run this case study on five machines, we use *Amazon Web Services* (AWS),[3] which is a collection of WSs that together make up a Cloud computing platform, offered over the Internet by Amazon.com. In particular, we take advantage of *Elastic Compute Cloud* (EC2): a WS that provides resizable compute capacity in the Cloud. It is designed to make Web-scale computing easier for developers, by allowing them to rent virtual machines on which to run their own applications. We rented five of those virtual

[3] http://aws.amazon.com.

machines and deployed the generated code for this case study as explained above.

By using the Cloud in this way, we can position the example in [37] in the following new scenario. Suppose we run a (large) online business company that offers a purchase service through AWS: the motivation is that our company desires to scale its business in the Cloud, benefiting from a third-party infrastructure that can efficiently manage large amounts of data. For instance, combined with EC2, our company can use another Cloud service from the AWS platform called *Simple Storage Service* (S3): an online technology for managing large amounts of information at any time, such as transaction logs of our clients' orders. Because the hybrid approach allows us to sensibly distribute CA-implementations over machines (thereby minimizing network traffic, as explained above), it is—in contrast to the purely centralized approach—more suitable for coordination of large amounts of data.

6 Performance comparison

In the introduction of this paper, we stated that the hybrid approach should strike a middle ground between latency and parallelism at run time while achieving reasonably fast compilation at build time. Having worked out the design and the implementation of a theory to automatically construct hybrid connector implementations in Sects. 3 and 4 and having illustrated the new opportunities that this technology provides in Sect. 5, we present a first preliminary performance comparison in this section to explore whether the hybrid approach indeed lives up to our expectations in the introduction.

6.1 Setup

To make our comparison, we use two Reo-to-Java compilers: a centralized-approach compiler and a "sibling" of the hybrid-approach compiler of Sect. 4. The latter works according to the same principles as the one in Sect. 4, but it generates code optimized for shared-memory deployment. By focusing on shared memory instead of distributed memory in our comparison (contrasting our case study in Sect. 3), we can more faithfully measure the intrinsic performance of generated centralized and hybrid connector implementations, without our measurements becoming obscured or even dominated by network latency. Also, for this comparison, we compiled connectors in a data-unaware fashion (i.e., generated connector implementations do not solve data constraints at run time but simply assume them to hold true), thereby excluding the time spent on solving data constraints from our measurements. Otherwise, because solving data constraints has a significant impact on performance, our measurements would have become dominated by irrelevant—to this comparison—activities. After all, we primarily want to quantify the effect of our partitioning algorithm on run-time performance, which does not depend on data constraints, nor on network latency.

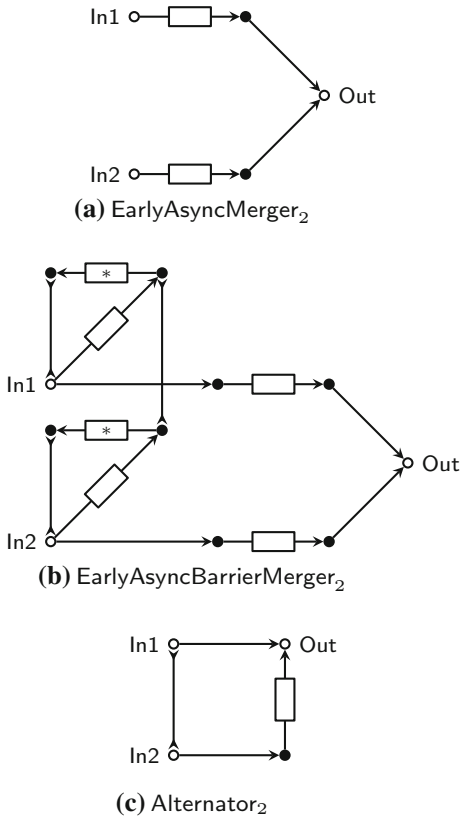


Fig. 15 Example connectors from the k-producers-single-consumer category

We consider two categories of connectors: *k-producers-single-consumer* and *single-producer-k-consumers*. In our comparison, both of these categories consist of three k-parametrized connector families for which we generated centralized and hybrid implementations for $k \in \{2, 4, 6, 8, 10, 12, 14, 16, 32, 48, 64\}$. In total, thus, we considered 66 different connectors and twice as many implementations. We ran every generated connector implementation nine times on a machine with sixteen cores (two Intel E5-26520V2 processors with eight physical cores at 2.6 GHz in two sockets, hyperthreading disabled) and averaged our measurements. In every run, we warmed up the JVM for 30 s before actually starting to measure the number of "rounds" that a connector implementation could finish in the subsequent 4 min. What constitutes one round differs per connector; we provide details below. Neither the producers nor the consumers performed any computation to focus our comparison on the performance of the generated connector implementations.

6.2 Connectors

Figure 15 shows three of the connectors in the k-producers-single-consumer category.

Figure 15a shows EarlyAsyncMerger$_2$ (the $k = 2$ version of EarlyAsyncMerger$_k$). With this connector, whenever a producer sends a data item through its local port, the connector stores this data item in a corresponding FIFO buffer. The producer can immediately continue, possibly before the consumer has received (i.e., communication between a producer and the consumer transpires asynchronously). Whenever the consumer receives a data item through its local port, the connector empties one of the previously full FIFO buffers, selected nondeterministically. The consumer does not necessarily receive data items in the order in which producers sent them (i.e., communication between a producer and the consumer transpires not necessarily transactionally). Every round consists of a send by a producer and a receive by the consumer.

Figure 15b shows EarlyAsyncBarrierMerger$_2$ (the $k = 2$ version of EarlyAsyncBarrierMerger$_k$). This connector works in largely the same way as EarlyAsyncMerger$_2$, except that EarlyAsyncBarrierMerger$_2$ enforces a barrier on the producers: no producer can send its ith data item until all the other producers have sent their $(i - 1)$th data items. As with EarlyAsyncMerger, the consumer may still receive data items in an order different from the order in which the producers sent them. Every round consists of a send by every producer and k receives by the consumer, one for every producer.

Figure 15c shows Alternator$_k$ (the $k = 2$ version of Alternator$_k$, previously shown in Fig. 1). With this connector, whenever a producer (attempts to) sends a data item through its local port, it blocks both until the consumer (attempts to) receives a data item through its local port and until every other producer (attempts to) sends a data item through its local port (i.e., the producers can send only synchronously). Once each of the producers and the consumer (attempt to) send/receive, the consumer receives the data item sent by the top producer (i.e., communication between the top producer and the consumer transpires synchronously), while the connector stores the data items of the other producers in their corresponding FIFO buffers (i.e., communication between the other producers and the consumer transpires asynchronously). Afterward, the consumer receives the other buffered data items in the top-to-bottom order in which the producers are arranged. Every round consists of a send by every producer and k receives by the consumer, one for every producer.

Figure 16 shows three of the connectors in the single-producer-k-consumers category.

Figure 16a shows LateAsyncReplicator$_2$ (the $k = 2$ version of LateAsyncReplicator$_k$). With this connector, whenever a producer sends a data item through its local port, the connector stores (copies of) this data item in every FIFO buffer. The producer can immediately continue, possibly before the consumers have received (i.e., communication between the producers and a consumer transpires asynchronously). Whenever a consumer receives a data item through its local port, the connector empties its corresponding full FIFO buffer. Every round consists of a send by the producer and k receives by the consumers, one by every consumer.

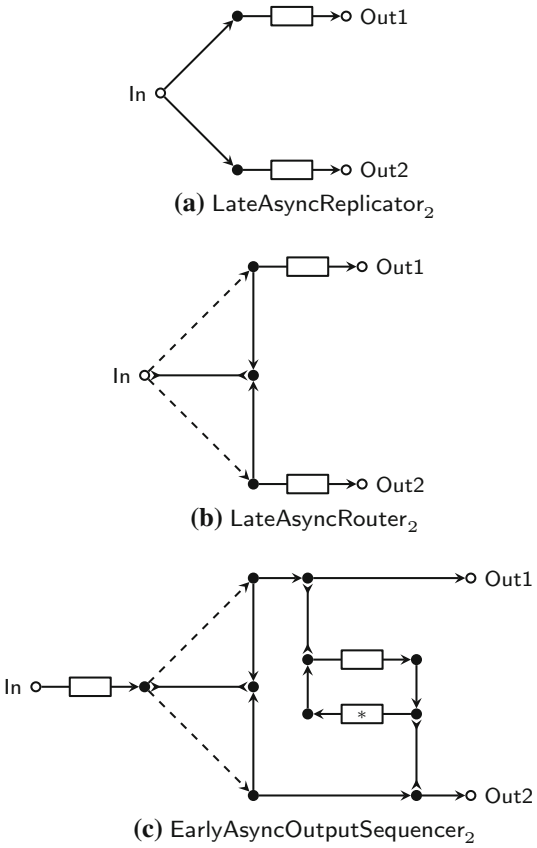


Fig. 16 Example connectors from the single-producer-k-consumers category

Figure 16b shows LateAsyncRouter$_2$ (the $k = 2$ version of LateAsyncRouter$_k$). With this connector, whenever the producer sends a data item through its local port, the connector stores this data item in exactly one of the FIFO buffers (instead of in all buffers as LateAsyncReplicator$_k$ does), selected nondeterministically. The producer can immediately continue, possibly before the consumer of the selected buffer has received (i.e., communication between the producer and a consumer transpires asynchronously). Whenever a consumer receives a data item through its local port, the connector empties its corresponding full FIFO buffer. The consumers do not necessarily receive data items in the order in which the connector stored those data items in the FIFO buffers. Every round consists of a send by the producer and a receive by a consumer.

Figure 16c shows EarlyAsyncOutputSequencer$_2$ (the $k = 2$ version of EarlyAsyncOutputSequencer$_k$). With this connector, whenever the producer sends a data item through its local port, the connector stores this data item in

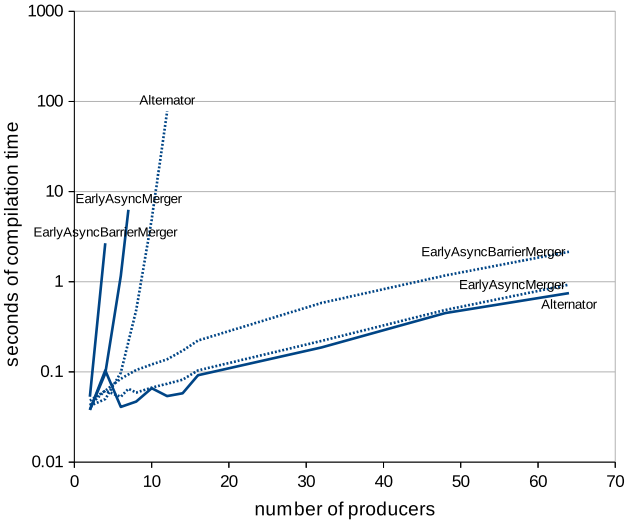


Fig. 17 Compilation time of k-producers-single-consumer connectors (*solid lines* for centralized-approach compiler; *dotted lines* for hybrid-approach compiler)

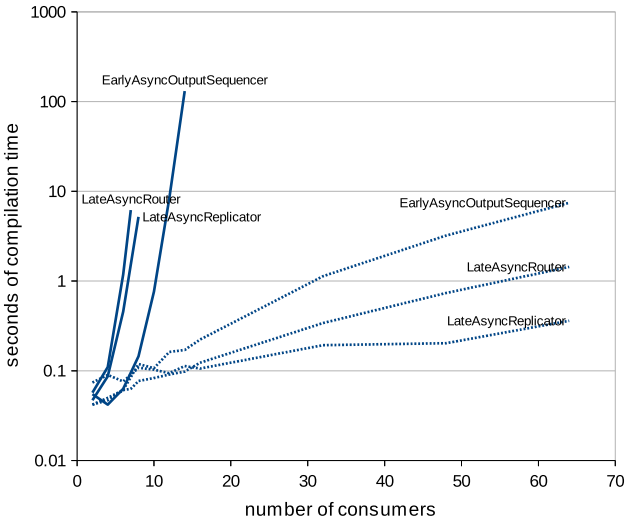


Fig. 18 Compilation time of single-producer-k-consumers connectors (*solid lines* for centralized-approach compiler; *dotted lines* for hybrid-approach compiler)

the leftmost FIFO buffer. The producer can immediately continue, possibly before the consumer has received (i.e., communication between a producer and the consumers transpires asynchronously). The connector ensures that the consumers can receive only in sequence (from top to bottom). Whenever a consumer receives a data item through its local port, the connector empties its corresponding full FIFO buffer. Every round consists of k sends by the producer and k receives by the consumers, one by every consumer.

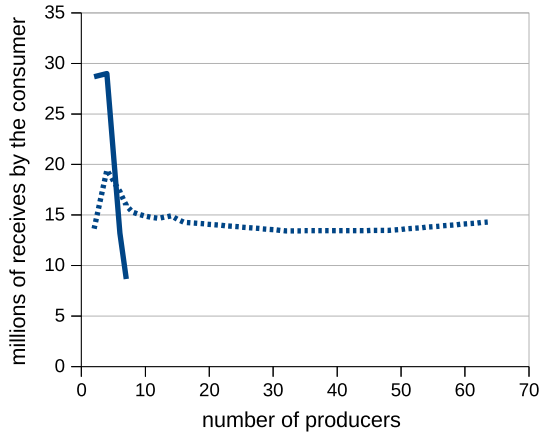
6.3 Results

Figures 17 and 18 show the compilation times of the connectors in Figs. 15 and 16 for the aforementioned values for k. Note the logarithmic scale on the vertical axis. For all but one connector family, the compilation time of the hybrid-approach compiler is much lower than that of the centralized-approach compiler. In fact, for five of the six connector families, the centralized-approach compiler failed to run to completion beyond certain (relatively low) values of k:

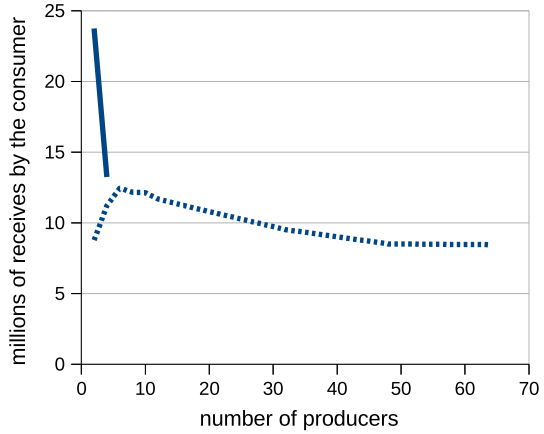
- For $\mathsf{EarlyAsyncMerger}_{k>7}$, the number of transitions in the computed "big" CA exceeded the limit of 2048 transitions (e.g., $\mathsf{EarlyAsyncMerger}_8$ has 23801 transitions). We imposed this transition limit, because (i) crossing this relatively high limit with relatively low number of producers signifies exponential growth and (ii) the Java compiler cannot conveniently handle Java code generated for automata with so many transitions.
- For $\mathsf{EarlyAsyncBarrierMerger}_{k>4}$, the number of transitions in its big CA exceeded the limit of 2048 (e.g., $\mathsf{EarlyAsyncBarrierMerger}_5$ has 10,943 transitions).
- For $\mathsf{LateAsyncReplicator}_{k>8}$, the number of transitions in its big CA exceeded the limit of 2048 (e.g., $\mathsf{LateAsyncReplicator}_9$ has 19,172 transitions).
- For $\mathsf{LateAsyncRouter}_{k>7}$, the number of transitions in its big CA exceeded the limit of 2048 (e.g., $\mathsf{LateAsyncRouter}_8$ has 23,801 transitions).
- For $\mathsf{EarlyAsyncOutputSequencer}_{k>16}$, the compiler crashed with an `OutOfMemoryError`.

The previous observations for these five connector families are witnessed also by their very steep curves in Figs. 17 and 18 (i.e., all solid lines except the one for Alternator), which indicate poor scalability in terms of compilation performance. Only for $\mathsf{Alternator}_k$, the centralized-approach compiler succeeded in generating code for all of our values of k within a reasonable time frame. Interestingly, for the hybrid-approach compiler, the exact opposite holds: while it easily succeeded in generating code for five out of six connector families, it failed for $\mathsf{Alternator}_{k>12}$. In the next subsection, we discuss what makes $\mathsf{Alternator}_k$ special.

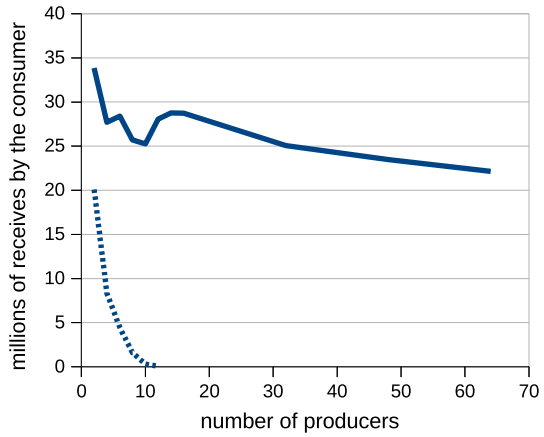
Figure 19 shows the execution times of the connector families in the k-producers-single-consumer category, averaged over nine runs. For $\mathsf{EarlyAsyncMerger}_k$ and $\mathsf{EarlyAsyncBarrierMerger}_k$, their centralized implementations outperform their hybrid implementations in cases involving only few producers (six in the case of $\mathsf{EarlyAsyncMerger}_k$; four in the case of $\mathsf{EarlyAsyncBarrierMerger}_k$). In cases involving more producers, either the hybrid implementations outperform the centralized implementations, or the centralized-approach compiler failed to compile in which case we cannot make a comparison. In those latter cases, however, it seems reasonable to assert by extrapolation that if compilation had succeeded, these generated centralized implementations would have performed significantly



(a) EarlyAsyncMerger$_k$

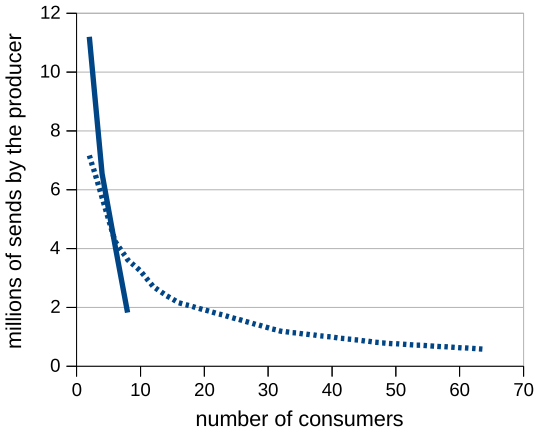


(b) EarlyAsyncBarrierMerger$_k$

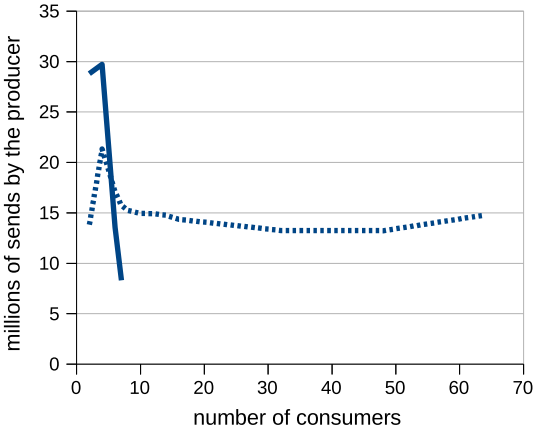


(c) Alternator$_k$

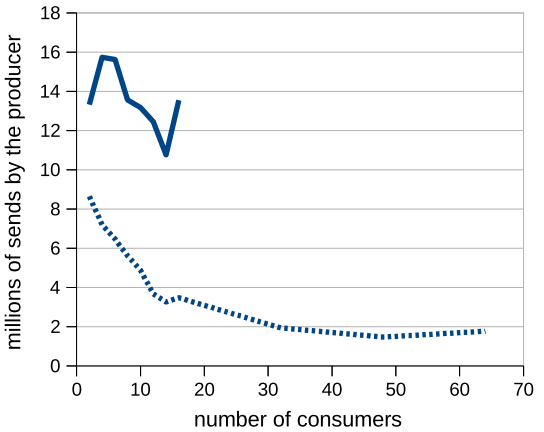
Fig. 19 k-producers-single-consumer (*solid/dotted line* for centralized/hybrid implementation)



(a) LateAsyncReplicator$_k$



(b) LateAsyncRouter$_k$



(c) EarlyAsyncOutputSequencer$_k$

Fig. 20 k-producers-single-consumer (*solid/dotted line* for centralized/hybrid implementation)

worse than their corresponding hybrid implementations. For Alternator$_k$, in contrast, its centralized implementations outperform its hybrid implementations. We discuss this phenomenon in more detail in the next subsection.

Figure 20 shows the execution times of the connector families in the single-producer-k-consumers category, averaged over nine runs. The figures for LateAsyncReplicator$_k$ and LateAsyncRouter$_k$ are similar to those of EarlyAsyncMerger$_k$ and EarlyAsyncBarrierMerger$_k$ that we saw before: with only few consumers, the centralized implementations outperform the hybrid implementations, while with more consumers, the hybrid implementations outperform

the centralized implementations. For EarlyAsyncOutputSequencer$_k$, since the centralized-approach compiler failed to generate code for $k > 16$, the comparison remains inconclusive. We discuss this case in more detail in the next subsection.

6.4 Discussion

For four of the six connector families under investigation, the hybrid approach has clear advantages over the centralized approach. In each of those cases, the hybrid approach scales much better than the centralized approach both in terms of compilation performance at build time and latency at run time. In one of the remaining cases, however, the centralized approach scales better than the hybrid approach; in the other remaining case, our comparison seems inconclusive. We analyze those two cases below.

For Alternator$_k$, the hybrid approach has two issues that the centralized approach has not: *transition relation explosion* at build time and *oversequentialization* at run time. Jongmans and Arbab recently discovered these issues and explained them in detail [32]; below, we summarize their findings.

The partition for Alternator$_k$ as computed by the hybrid-approach compiler consists of k parts: $k-1$ singleton subsets, each of which consists of one FIFO primitive, and one large subset A, which consists of all the other primitives. For each of the FIFO primitives except the one at the top, product CA $[\![A]\!]$ has a transition for propagating a data item from the buffer of that FIFO primitive into the buffer of the FIFO primitive directly above it. This results in $k-2$ "buffer propagation"-transitions. By true concurrency, however, any subset of those transitions may in principle occur simultaneously. Each of those subsets manifests in $[\![A]\!]$ as a separate transition. Thus, $[\![A]\!]$ has at least 2^{k-2} transitions. This exponential increase in transitions as k increases causes severe *transition relation explosion* at build time, as also shown in Fig. 17. The centralized-approach compiler, in contrast, does not suffer from transition relation explosion. This has two reasons.

1. Even though any subset of "buffer propagation"-transitions may *in principle* occur simultaneously, the emptiness/fullness of buffers involved in those transitions makes many of those transitions permanently disabled in practice. Whether or not a transition is permanently disabled at run time can be established at build time only by computing the full product of $[\![A]\!]$ and every FIFO primitive. By doing so, the FIFOs effectively constrain which transitions of $[\![A]\!]$ actually can occur. The centralized-approach compiler computes this full product and as such eliminates many impossible "buffer propagation"-transitions; the hybrid-approach compiler does not.
2. The centralized-approach compiler can perform another operation on CA, called *hide* [4], to abstract away internal transitions in the full product, including the remaining "buffer propagation" transitions. The hybrid-approach compiler, in contrast, cannot abstract those "internal" transitions away, as it cannot treat such "buffer propagation" transitions as truly internal: these transitions involve ports that lie on the boundary between CA in different subsets in the partition and therefore must remain.

For these reasons, the centralized approach has only a linear increase in transitions as k increases instead of exponential.

The previous discussion explains why the centralized-approach compiler succeeds in generating code for Alternator$_k$, while the hybrid-approach compiler fails for $k > 12$. However, this does not yet explain why also at run time, centralized implementations of Alternator$_k$ outperform their hybrid implementations. The reason becomes clear when we realize that Alternator$_k$ essentially has purely sequential behavior: in every round, the producers start by synchronously sending their data items (and the consumer synchronously receives the first data item), after which the consumer receives the remaining $k-1$ data items in sequence. The centralized implementation of Alternator$_k$ at run time sequentially simulates one CA consisting of k transitions between k states that represents exactly this sequentiality. The hybrid implementation, in contrast, at run time has k parallel CA-implementations (i.e., one for the large subset in the partition and $k-1$ for every singleton) and, thus, suffers from *overparallelization*: it uses parallelism—and incurs the overhead that parallelism involves—to implement intrinsically sequential behavior.

The Alternator family shows that the hybrid approach as presented in the previous sections of this paper is not the end of the story. Indeed, the hybrid approach does not work in all cases. In future work, we need to develop better static analysis techniques to equip our compilers with. Based on such techniques, a compiler must be able to determine whether it should use a centralized compilation approach or a hybrid one. A first naive heuristic may be to merge a subset A_1 with a subset A_2 in a computed partition if all CA in A_1 have direct neighbors only in A_2 or vice versa. Applied to Alternator$_k$, this heuristic effectively reduces the hybrid approach to the centralized approach, which in this case is what we want. However, we need to study this heuristic in much more detail to check whether it does not have problematic consequences. Also, this heuristic does not cover all cases. For instance, EarlyAsyncOutputSequencer$_k$—the other case in which the centralized approach seems to outperform the hybrid approach—also has intrinsically sequential behavior, but the previous naive heuristic does not work here. In fact, the centralized approach in its present form does not help much here either, because the centralized-approach compiler crashed on

an `OutOfMemoryError` for $k > 16$ as shown in Fig. 18. To handle $\mathsf{EarlyAsyncOutputSequencer}_k$ in a scalable manner, thus, much more advanced compilation techniques seem necessary.

In this section, we tried to provide a balanced perspective on the performance of the hybrid approach in practice. Based on this first comparison, we can draw the following preliminary conclusion: if the hybrid approach works, it works well. This means that the hybrid approach indeed balances latency and parallelism to get better performance than the purely centralized approach at run time while achieving reasonably fast compilation at build time. Additionally, however, our comparison reveals limitations of the hybrid approach. In future work, we intend to refine the hybrid approach (to handle $\mathsf{Alternator}_k$-like connectors) and develop complementary technology (to handle $\mathsf{EarlyAsyncOutputSequencer}_k$-like connectors).

Recently, we thoroughly repeated and extended the experiments reported in this paper on a machine with 24 cores (two Intel E5-2690V3 processors with twelve physical cores at 2.6, GHz in two sockets, hyperthreading disabled). We report on these new experiments elsewhere [33]. An important difference between our new experiments and the experiments reported in this paper is that we statically fixed the clock frequency in our new experiments, thereby effectively disabling Turbo Boost. On the one hand, disabling Turbo Boost makes the results of our new experiments "purer" than the ones reported in this paper (where the system did not allow disabling Turbo Boost). On the other hand, the Turbo Boost disabled environment of our new experiments is more synthetic (because most modern processors support Turbo-Boost-like technology), which makes the results reported in this paper (with Turbo Boost enabled) more realistic than the "purer" results of our new experiments. Of course, the absolute performance numbers obtained in our two sets of experiments differ. Significantly, however, the general shapes of the graphs and the trends that we observe in the results of the two sets of experiments are similar, modulo the local perturbations in the graphs presented in this paper, which may show the effect of Turbo Boost.

7 Related work

In this section, we discuss related work on Reo, distributed coordination, and distributed orchestration/workflow.

7.1 Reo

Closest to ours is the work on splitting connectors into (a)synchronous regions for better performance. For his PhD thesis [60], Proença developed the first implementation based on these ideas, demonstrated its merit through benchmarks, and invented a new automaton model to reason about split connectors [58,59]. Furthermore, Clarke and Proença explored connector splitting in the context of the connector coloring semantics [18]. 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. [34] studied a formal justification of connector splitting in a process algebraic setting.

Also related to the work presented in this paper is the work of Kokash et al. on *action constraint automata* (ACA) [40]. Kokash et al. argue that ordinary constraint automata describe the behavior of Reo connectors too coarsely, which makes it impossible to express certain fine parallel behavior. While ACA better describe the behavior of existing connector implementations (under certain assumptions), the increased granularity of ACA comes at the price of substantially larger models. This makes them less suitable for code generation.

7.2 Distributed coordination

We proceed with a discussion of related work on other approaches to distributed coordination.

For instance, Rowstron and Hood [63] present a new set of primitives for fully distributed coordination of processes using tuple spaces, called the *Bonita* primitives. Whereas the original *LINDA* primitives provide only synchronous access to tuple spaces, the Bonita primitives provide asynchronous access, allowing processes to perform computations concurrently with tuple space operations (thus, increasing performance). These new primitives are *dispatch*, *dispatch_bulk*, *arrived*, and *obtain*, which are all designed to access distributed tuple spaces. Moreover, more efficient coordination constructs can be produced using these primitives, as an *ALT* construct, which allows a number of different tuples to be requested and then perform actions as the result arrives.

A further main reference in tuple-based distributed coordination is represented by the *Kernel Language for Agent Interaction and Mobility* (*KLAIM*) [52]. This language consists of a core LINDA with multiple tuple spaces and of a set of operators for building processes. KLAIM naturally supports programming with explicit localities. Localities are first-class data, i.e., they can be manipulated like any other data, and the provided mechanisms control the interactions among located processes. Moreover, KLAIM uses a type system that statically checks access rights violations of mobile agents.

A more recent proposal by some of the same authors of [52] is *Service Component Ensemble Language* (*SCEL*) [20],

a new language specifically designed to model autonomic components and their interaction. It brings together various programming abstractions that permit to directly represent knowledge, behaviors, and aggregations according to specific policies. It also supports naturally programming self-awareness, context-awareness, and adaptation. Some syntactic categories are left open to represent, for instance, knowledge of different forms (e.g., constraints, clauses, records, tuples) or to express a variety of policies (e.g., to regulate knowledge handling, resource usage, process execution, process interaction, actions priority, security, trust, reputation).

One more main work, still based on LINDA, is *Linda In a Mobile Environment* (*LIME*) [49], which is a model and middleware that supports development of applications that exhibit physical mobility of hosts, logical mobility of agents, or both. The set of tuples being shared changes over time as a result of the agents' local control regarding sharing, and in response to the mobility of both agents and hosts. When hosts come within each others' communication range, the set of shared tuple spaces expands, and when they move apart, it contracts. LIME represents a thorough approach, since it comes with a formal semantic definition, implementation pragmatics (for instance, there is a Java implementation), and application-driven evaluation of the resulting model and middleware.

Modeling *Mobile Computing* with process calculi is a popular topic in the literature since the 1990s. Besides KLAIM, examples include, to name a few, the *Ambient Calculus* [15] (an ambient is informally defined as a bounded place in which computation can occur), a mobile extension of *Concurrent Constraint Programming* [26], and the *Join-Calculus* [23].

In [61], the authors present a coordination model, the *Actor, Role, and Coordinator* (*ARC*) model, to address three main concerns inherent in a pervasive *Open Distributed and Embedded* (*ODE*) system: *dynamicity*, *scalability*, and stringent *QoS requirements*. The model treats a pervasive ODE system as a composition of concurrent computation and coerced coordination. Concurrent computation is modeled as actors, while coerced coordination describes the system's QoS requirements by mapping them to coordination constraints. The coordinators are responsible for the coordination among roles, while the roles in the model provide abstractions for coordinated behaviors that may be shared by multiple actors and further assume local coordination responsibilities for the actors playing the roles. The role's behavior abstraction decouples the syntactic dependencies between the coordinators and the actors, thus shielding the coordinator layer from the dynamicity of underlying actors.

A different approach is represented by *Event-Based Systems* (*EBS*) [46]. In an event-based mode of interaction, components communicate by generating and receiving event notifications, where an event is any occurrence of a happening of interest, i.e., a state change in some component. The affected component issues a notification describing the observed event. An event notification service or publish/subscribe middleware mediates between the components of an EBS and conveys notifications from producers (or publishers) to consumers (or subscribers) that have registered their interest with a previously issued subscription.

Besides previous models and languages, in the literature we can find nature-inspired models, such as the *Gamma* model [5] (and its distributed implementations), where a shared coordination space is ruled by chemical-like laws defined by the programmers, globally working like a rewriting system; thus, it resembles the features of programmable tuple space models. Moreover, we have field-based coordination models [45] that are inspired by the way masses and particles move and self-organize according to gravitational or electromagnetic fields. Typically, a pervasive coordination infrastructure (of multi-agents) generates and maintains computational force fields, which are sensed and modified by agents moving through the fields, according to the field intensity. In *"Tuples on the Air"* (*TOTA*) [44], computational force fields take the form of distributed tuples, which are generated both by the active components and by the pervasive coordination infrastructure, and drive the actions and motion of the component themselves; for instance, they allow agents to find each other in a dynamic network.

The *stigmergy* paradigm has been adopted in multi-agent systems and in other fields as a technique for realizing forms of emergent coordination in societies composed by a large amount of ant-like, nonrational agents [21]. The principle is that the trace left in the environment by an action stimulates the performance of a next action, by the same or a different agent (typical of *self-organizing systems*). In [62], the authors generalize it to *cognitive stigmergy*, in order to enable social activities of cognitive agents.

The Reo language proves to have a completely different approach from all the references previously summarized; for instance, it is not tuple-based as are all the derivatives of LINDA, it does not resemble a process calculus, and it is not directly environment-driven (as with the stigmergy paradigm). Nevertheless, the Reo language has a strong formal basis and promotes loose coupling, distribution, mobility, exogenous coordination, user-defined primitives, arbitrary mix of synchrony and asynchrony, and dynamic reconfigurability. It has a formal graphical syntax (analogous to electronic circuit diagrams), and a few formal semantics based, for instance, on compositional CA [4]. The formal basis of Reo guarantees possibilities for both model checking and verification [41].

Finally, Bonakdarpour et al. [11] present an approach for generating distributed implementations for specifications in BIP [7], 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) CA, and the middle layer of execution engines roughly coincides with our set of "medium" CA-implementations. 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 having the partition algorithm in Fig. 7 put CA with "conflicting transitions" in the same part, 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.

7.3 Distributed orchestration/workflow

To put our previous case study in perspective, we conclude with related work on distributed orchestration/workflow.

Nanda et al. [51] present a technique for partitioning a composite service written as a single BPEL [39] program into an equivalent set of decentralized processes. In their approach, Nanda et al. construct the dependency graph of a BPEL program with the aim of minimizing communication costs while maximizing throughput. Chafle et al. [16] decentralize the orchestration of a *FindRoute* service by partitioning the BPEL code into four parts, which are executed by four distinct Java engines. Afterward, Chafle et al. compare the performance of the centralized and the decentralized implementation by using both service-time and message-size metrics. Chafle et al. also estimate the additional complexity in error recovery and fault handling in a decentralized orchestration. Mostarda et al. [48] use a BPEL-based language for distributed orchestration in the context of pervasive computing. In their approach, Mostarda et al. automatically decompose a centralized workflow into implementations of finite state machines that, when synchronized using a consensus protocol, execute the original workflow. Fernández et al. [22] introduce an execution model for distributed orchestration based on a metaphor from chemistry. In this approach, services communicate with each other through a distributed shared multiset ("chemical solution"), which contains both control-flow and data-flow information ("molecules"). A workflow engine then executes an orchestration protocol by applying formal rewrite rules ("chemical reactions"). Fernández et al. propose translating BPEL programs to chemical representations that the workflow engine they describe can process. The difference between these existing BPEL-based approaches to distributed orchestration and our approach is the use of Reo instead of BPEL. At least from a software engineering perspective, Reo has several advantages compared to BPEL (declarative style, verbatim reuse/composition of connectors, smaller gap between verification and execution; see also Sect. 1). Whether our Reo-based approach also outperforms existing BPEL-based approaches remains a topic for future study.

Barker et al. [6] presents an architecture (including a proxy API) for optimizing data flow in workflow execution. In their architecture, control-flow messages are still sent to a centralized orchestration engine, while data-flow messages (i.e., the actual, potentially large, pieces of data) go directly from one service to another. The latter reduces network traffic, and it coincides with the heuristic we adopted for distributing the CA-implementations over machines in our case study. However, our technique distributes also control flow. This gives our approach, in addition to data-flow optimization through clever deployment, also other advantages such as control-flow parallelism, no single point-of-failure, and potentially easier dynamic reconfiguration.

Similarly, motivated as the work of Barker et al. (i.e., optimizing data-flow), Binder et al. [10] propose a distributed orchestration methodology based on decomposing an orchestration protocol (or workflow) into a directed acyclic graph of service invocations, represented as *triggers*. Every trigger encodes the data dependencies of the invocation it represents (i.e., the parents and children of the corresponding node in the graph). Triggers act as proxies: they collect all input data before actually invoking the service, and they transmit all output data directly to the dependent triggers for subsequent service invocations. One may regard the medium CA-implementations that we generate with our code generator as triggers. In that case, the work presented in this paper improves the work of Binder et al. by presenting an automatic procedure for decomposing orchestration protocols into triggers for a more expressive protocol language (i.e., Reo): we support also loops and conditions, which Binder et al. forbid.

In [66], Tretola and Zimeo present a technique for improving concurrency in execution of workflows. Their technique works by concurrently invoking otherwise sequential, data-dependent services with pieces of placeholder data until those services require the actual data. If service invocations have a data-independent initialization phase, for instance, the work-

flow engine of Tretola and Zimeo executes those initialization phases in parallel instead of in sequence. For workflows and services implemented on top of the framework of Tretola and Zimeo, this process happens automatically. Although Tretola and Zimeo parallelize service invocations, they neither parallelize nor distribute the execution of the orchestration protocol. Also, their technique seems unable to handle third-party/black-box services. In contrast, our approach does distribute the execution of the orchestration protocol and works with black-box services (via proxies [37]), but we currently cannot exploit potential concurrency between service invocations the way Tretola and Zimeo do. Thus, the technique of Tretola and Zimeo seems to complement our work. (However, we do not understand yet which notion of equivalence their technique preserves and what consequences this has for properties that the orchestration protocol satisfies on the modeling level. This requires further study.)

Pedraza and Estublier [56] present FOCAS. This framework consumes as input an annotated APEL [19] specification of an orchestration scenario and produces as output a number of suborchestrations and a deployment plan (for distributing suborchestrations over machines). The main difference between our approach and the approach of Pedraza and Estublier seems to be that Reo allows for expressing more complex data-flow behavior than APEL does. This directly influences the complexity of automatically computing suborchestrations. On the other hand, our code generator currently does not compute a deployment plan, because we do not support deployment annotations, which FOCAS does.

Finally, Muth et al. [50] use the formal semantics of state/activity charts to develop an algorithm for transforming centralized state/activity charts into equivalent partitioned ones, suitable for distributed execution by a workflow engine. Muth et al. subsequently refine their basic approach to also reducing communication overhead and exploiting parallelism between parts in partitions. We had similar motivations for doing the work presented in this paper, but our formalism differs. Comparing the strengths and weaknesses of state/activity charts and Reo/CA in the context of workflows, orchestration protocols, and general coordination seems an interesting topic for future work.

8 Discussion: applications beyond Reo

Because we formulated our results generally in terms of constraint automata (i.e., Reo's semantics, essentially independent of Reo), any system expressible in terms of such automata can benefit from these results. To illustrate this point, in this section, we sketch two possible applications beyond Reo. The purpose of this section is not to exhaustively present new applications, but it serves as an overview of two possible pieces of future work.

8.1 Automatic code generation for Rebeca models

Rebeca is an actor-based language for modeling and verifying concurrent and distributed systems [65]. Every *Rebeca model* consists of a number of *rebecs* (cf. actors), each of which has its own memory and its own single thread of execution. Rebecs communicate with each other through asynchronous message passing. On receiving a message, a rebec executes the body of the method referred to in that message. As part of executing a method, a rebec may asynchronously send messages to other rebecs. Every rebec has an unbounded message queue to buffer incoming messages. Rebeca's semantics states that methods execute atomically. This means that a scheduler can schedule only one rebec for execution at a time (i.e., rebecs execute concurrently but not truly parallel).

Rebeca's semantics can be expressed compositionally as a CA product [64]. More precisely, the set of CA that constitute the CA product for a Rebeca model can be split into four sections: (i) the first section contains, for every rebec, one CA that models its local behavior, (ii) the second section contains, for every rebec, one CA that models its unbounded message queue, (iii) the third section contains a number of CA for scheduling rebecs, and (iv) the fourth section contains a number of CA for routing messages between rebecs. (To model an unbounded message queue with a finite CA, we must assume that our data domain contains an unbounded data structure. In that way, we can construct a CA with a memory cell containing that data structure to encode an unbounded queue and use that CA's transitions to encode operations on that queue. Alternatively, for bounded queues, we can use a FIFO-like CA without making assumptions about the data domain.)

Because Rebeca models can be expressed as products of CA, even our existing centralized-approach compiler already facilitates automatically generating code for verifiably correct Rebeca models. Although this use of Rebeca goes beyond its original intent, such code generation enables a useful Rebeca-based correct-by-construction design methodology. However, centralized-approach implementations of Rebeca models have two disadvantages. First, the "big" CA for a Rebeca model expressed as a product of CA has a number of states exponential in the number of rebecs. Consequently, our centralized-approach compiler can compile only relatively small Rebeca models within a reasonable time limit. Second, even if our centralized-approach compiler succeeds in compiling a sufficiently small Rebeca model, all rebecs in the resulting implementation must run on the same physical machine. This makes it impossible to generate distributed systems from Rebeca models (i.e., even if rebecs are distributed at the modeling level, the generated implementation deficiently inhibits their actual distribution in practice).

The technique presented in this paper allows us to partition the CA product for a Rebeca model into parts and

generate code on a per-part basis, as follows. Every rebec communicates only asynchronously and therefore satisfies no-synchronization. Consequently, the CA for every rebec has its own part in the partition. Similarly, the message queue of every rebec has its own part in the partition. The CA in the third section (scheduling rebecs) are all independent of the CA in the fourth section (routing messages), and so, these sections constitute two separate parts in the partition. Thus, the CA product for a Rebeca model consisting of k rebecs can be partitioned into $2k + 2$ parts. The subsequently generated implementation has one CA-implementation for every rebec, one CA-implementation for every message queue, one CA-implementation for scheduling rebecs, and one CA-implementation for routing messages. This approach palliates exponential state explosion, and it allows (CA-implementations for) rebecs to be distributed across different machines.

8.2 Recovering projectable choreographies from unprojectable specifications

Another possible application of our results beyond Reo is *projection* in choreography languages [12–14,24,25,27]. A projection maps a global protocol specification among k parties, called a *choreography*, to k local specifications of per-party observable behavior, called *contracts* [12,13] (also called *peers* [24,25] or *end-point processes* [14,27]). The challenge is to project a choreography in such a way that the collective behavior of the resulting contracts *conforms* with the projected choreography.

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 algorithmically inferring which communication actions need to be added to otherwise unprojectable choreographies (similar to the transformation by Lanese et al. [42]). Below, we present a first sketch.

Choreographies are commonly formally modeled as some kind of *labeled transition system* (LTS). To compute a projection of a choreography involving k parties, we take such an LTS as our starting point. If this LTS is finite, we translate it to a *choreography* CA (essentially by mapping transition labels in the LTS to sets of ports).[4] Afterward, we *decompose* the resulting "big" CA into a number of "small" CA using an existing decomposition algorithm [3]. Essentially, by recovering the internal structure of the big CA, this step reveals the previously "hidden" communication actions necessary to make the original choreography projectable. Next, we recombine the small CA into a number of *contract* CA such that for each of those CA, its input ports represent communication actions of only one party. To do this, we first apply the theory presented in this paper for computing a number of "medium" CA from the small CA. Subsequently, we take the product of every two medium CA whose ports belong to the same party, until no such CA exist anymore. Finally, we construct a number of sets of CA, each of which contains: (i) a contract CA resulting from the previous step and (ii) a number of Fifo CA such that the output port of every Fifo CA is the input port of the contract CA. Those Fifo CA essentially represent incoming message buffers of parties. The previous process yields one set of CA for every party. Every such set can then be compiled into the implementation of its corresponding party. Communication between CA of different sets (i.e., between different parties) has to satisfy only the local synchronization requirements imposed by $\boxdot$.

For instance, consider the following example of an unprojectable choreography (from Carbone et al. [14]):

$$\mathsf{Buyer} \rightarrow \mathsf{Seller}\ ;\ \mathsf{Shipper} \rightarrow \mathsf{Depot}$$

This choreography describes a sequence of two interactions. First, a party Buyer sends a message to another party Seller, presumably to buy some product. Subsequently, a third-party Shipper sends a message to a fourth party Depot, presumably to arrange the logistics of shipping the product sold by Seller to Buyer. Clearly, the interaction between Buyer and Seller *must* precede the interaction between Shipper and Depot for the whole protocol to make sense. However, it is impossible to directly project this choreography on the four parties in a way that satisfies this precedence. Basically, the choreography misses an essential interaction between Seller and Shipper, in which the former informs the latter about the sale. By extending the choreography as follows, thus, it becomes projectable [14]:

$$\mathsf{Buyer} \rightarrow \mathsf{Seller}\ ;\ \mathsf{Seller} \rightarrow \mathsf{Shipper}\ ;\ \mathsf{Shipper} \rightarrow \mathsf{Depot}$$

Next, we show that we can algorithmically derive a similar projection using the theory presented in this paper.

First, under the assumption that communication between parties transpires asynchronously, we model infinitely many repetitions of the initial choreography as the CA in Fig. 21. Ports $\mathsf{B}^{\mathrm{snd}}$, $\mathsf{Se}^{\mathrm{rcv}}$, $\mathsf{Sh}^{\mathrm{snd}}$, and $\mathsf{D}^{\mathrm{rcv}}$ denote the input port used

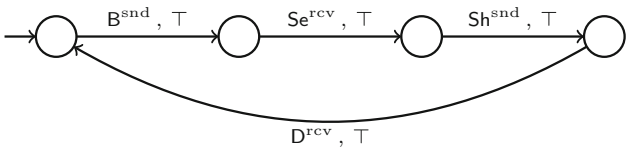


Fig. 21 Constraint automaton for unprojectable choreography Buyer → Seller ; Shipper → Depot

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

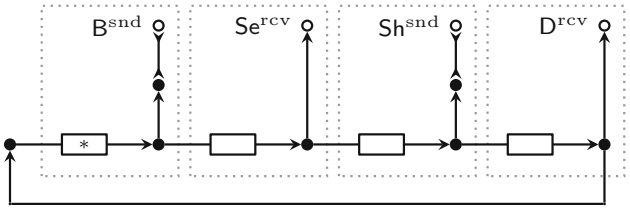


Fig. 22 Set of small CA, rendered as a Reo connector, obtained after decomposing the big CA in Fig. 21

by Buyer to send, the output port used by Seller to receive, the input port used by Shipper to send, and the output port used by Depot to receive, respectively. For simplicity, data constraint $\top$ means that we do not care about particular data in this example. Next, we decompose these CA into a number of smaller CA. Instead of showing the decomposition as a set of CA, we draw it as the Reo connector in Fig. 22. Now, we can partition this set of CA into eight disjoint subsets using the theory presented in this paper. Doing so yields one such subset for every FIFO and one for every vertical "pole" in Fig. 22. Subsequently, we merge the subset of every pole and the subset of its incoming FIFO into one subset, resulting in four subsets, represented by dotted rectangles in Fig. 22. Each of those subsets can now be compiled into the implementation of its corresponding party.

At run time, whenever Buyer sends a message (by performing a `write` on its input port B^{snd}), the CA compiled into Buyer's implementation also transfers a token from Buyer's incoming FIFO into the incoming FIFO of Seller. Observe that this interaction already occurred explicitly in the initial choreography. Subsequently, whenever Seller asynchronously receives that token (by performing a `take` on its output port Se^{rcv}), the CA compiled into Seller's implementation also transfers a copy of that token into the FIFO of Shipper. As such, it explicitly signals to Shipper that Shipper can start its interaction with Depot. The transfer of the token from Seller to Shipper corresponds to the third, additional interaction necessary to make the initial choreography projectable.

The sketched approach needs to be extended with data to be practically useful, but this example already illustrates the key ideas involved.

9 Conclusion

We presented a hybrid approach for the implementation of Reo connectors by partitioning them into several (a)synchronous regions at build time. Every such region can be executed on a different machine at run time. We use the term "hybrid" because our approach is neither purely centralized (regions run in parallel at run time), nor purely distributed (the elements inside a region are compiled to a sequential program at build time and require no distributed algorithms or communication at run time). In this way, we have the benefits of both pure approaches: the high run-time latency of a centralized scheme combined with the high parallelism and fast compilation of a distributed scheme.

As mentioned in Sect. 1, the value of our theoretical results extends beyond being an essential contribution to code generation technology for Reo, in at least two ways. First, because we formulated 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. This enables automatically generating hybrid implementations of systems specified in other languages. Second, our proof method, in which we compared 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 work by Lynch [43].

In the future, we plan to design an algorithm to automatically find the best partitioning of CA-implementations according to user and system-defined constraints. These constraints may need to be satisfied either "crisply" or "softly" (in relation to their indispensability) and may concern different criteria, including hardware requirements of software, QoS/QoE/performance desiderata, and issues correlated with security (e.g., preventing attacks based on *Business Process Discovery*), privacy (of both data and workflow), and fault handling.

References

1. Arbab F (2004) Reo: a channel-based coordination model for component composition. Math Struct Comput Sci 14(3):329–366. doi:10.1017/S0960129504004153
2. Arbab F (2011) Puff, the magic protocol. In: Agha G, Danvy O, Meseguer J (eds) Talcott Festschrift, LNCS, vol 7000. Springer, Berlin, pp 169–206. doi:10.1007/978-3-642-24933-4_9
3. Arbab F, Baier C, de Boer F, Rutten J, Sirjani M (2005) Synthesis of Reo circuits for implementation of component-connector automata specifications. In: Jacquet JM, Picco GP (eds) Proceedings of COORDINATION 2005, LNCS, vol 3454. Springer, Berlin, pp 236–251. doi:10.1007/11417019_16
4. Baier C, Sirjani M, Arbab F, Rutten J (2006) Modeling component connectors in Reo by constraint automata. Sci Comput Program 61(2):75–113. doi:10.1016/j.scico.2005.10.008
5. Banâtre JP, Fradet P, Métayer DL (2001) Gamma and the chemical reaction model: fifteen years after. In: Proceedings of the workshop on multiset processing: multiset processing, mathematical, computer science, and molecular computing points of view, WMP '00, pp 17–44. Springer, London, UK. http://dl.acm.org/citation.cfm?id=647269.721851

6. Barker A, Weissman J, van Hemert J (2008) Orchestrating data-centric workflows. In: Priol T, Jin H, Laforenza D, Matsuoka S, Parashar M, Roe P (eds) Proceedings of CCGRID 2008. IEEE, Los Alamitos, pp 210–217. doi:10.1109/CCGRID.2008.50
7. Basu A, Bozga M, Sifakis J (2006) Modeling heterogeneous real-time components in BIP. In: Hung DV, Pandya P (eds) Proceedings of SEFM 2006. IEEE, Los Alamitos, pp 3–12. doi:10.1109/SEFM.2006.27
8. ter Beek M, Bucchiarone A, Gnesi S (2004) Web service composition approaches: from industrial standards to formal methods. In: Galizia S, Emig C, Martens A, Roman D, Wombacher A (eds) Proceedings of ICIW 2007. IEEE, Los Alamitos, pp 224–233. doi:10.1109/ICIW.2007.71
9. Bernstein P, Hadzilacos V, Goodman N (1987) Two phase locking. In: Concurrency control and recovery in database systems. Addison-Wesley, Boston, pp 47–111
10. Binder W, Constantinescu I, Faltings B (2006) Decentralized orchestration of composite web services. In: Feig E, Zhang AKJ (eds) Proceedings of ICWS 2006. IEEE, Los Alamitos, pp 869–876. doi:10.1109/ICWS.2006.48
11. Bonakdarpour B, Bozga M, Jaber M, Quilbeuf J, Sifakis J (2012) A framework for automated distributed implementation of component-based models. Distrib Comput 25(5):383–409. doi:10.1007/s00446-012-0168-6
12. Bravetti M, Zavattaro G (2007) Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe M, Vanderperren W (eds) Proceedings of SC 2007, LNCS, vol 4829. Springer, Berlin, pp 34–50. doi:10.1007/978-3-540-77351-1_4
13. Bravetti M, Zavattaro G (2009) Contract compliance and choreography conformance in the presence of message queues. In: Bruni R, Wolf K (eds) Proceedings of WS-FM 2008, LNCS, vol 5387. Springer, Berlin, pp 37–45. doi:10.1007/978-3-642-01364-5_3
14. Carbone M, Honda K, Yoshida N (2012) Structured communication-centered programming for web services. ACM Trans Program Lang Syst 34(2):8:1–8:78. doi:10.1145/2220365.2220367
15. Cardelli L, Gordon A (1998) Mobile ambients. In: Nivat M (ed) Foundations of software science and computation structures, Lecture notes in computer science, vol 1378. Springer, Berlin, pp 140–155. doi:10.1007/BFb0053547
16. Chafle G, Chandra S, Mann V, Nanda MG (2004) Decentralized orchestration of composite web services. In: Najork M, Wills C (eds) Proceedings of WWW Alt. 2004. ACM, New York, pp 134–143. doi:10.1145/1013367.1013390
17. Changizi B (2015) Model based analysis of business process models. Ph.D. thesis, Leiden University (in preparation)
18. Clarke D, Proença J (2012) Partial connector colouring. In: Sirjani M (ed) Proceedings of COORDINATION 2012, LNCS, vol 7274. Springer, Berlin, pp 59–73. doi:10.1007/978-3-642-30829-1_5
19. Dami S, Estublier J, Amiour M (1998) APEL: a graphical yet executable formalism for process modeling. Autom Softw Eng 5:61–91. doi:10.1007/978-1-4615-5441-7_4
20. De Nicola R, Ferrari GL, Loreti M, Pugliese R (2011) A language-based approach to autonomic computing. In: Beckert B, Damiani F, de Boer FS, Bonsangue MM (eds) FMCO, vol 7542., Lecture notes in computer scienceSpringer, Berlin, pp 25–48
21. Dorigo M, Bonabeau E, Theraulaz G (2000) Ant algorithms and stigmergy. Future Gen Comput Syst 16(9):851–871. http://dl.acm.org/citation.cfm?id=348599.348601
22. Fernández H, Priol T, Tedeschi C (2010) Decentralized approach for execution of composite web services using the chemical paradigm. In: Pu C, Singhal S, Zhang J (eds) Proceedings of ICWS 2010. IEEE, Los Alamitos, pp 139–146. doi:10.1109/ICWS.2010.46
23. Fournet C, Gonthier G, Lévy JJ, Maranget L, Rémy D (1996) A calculus of mobile agents. In: Montanari U, Sassone V (eds) CONCUR, vol 1119., Lecture notes in computer scienceSpringer, Berlin, pp 406–421
24. Fu X, Bultan T, Su J (2004) Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor Comput Sci 328(1–2):19–37. doi:10.1016/j.tcs.2004.07.004
25. Fu X, Bultan T, Su J (2005) Realizability of conversation protocols with message contents. Int J Web Serv Res 2(4):68–93. doi:10.4018/jwsr.2005100104
26. Gilbert D, Palamidessi C (2000) Concurrent constraint programming with process mobility. In: Proceedings of the first international conference on computational logic, CL '00. Springer, London, UK, pp 463–477. http://dl.acm.org/citation.cfm?id=647482.728260
27. Honda K, Yoshida N, Carbone M (2008) Multiparty asynchronous session types. In: Necula G, Wadler P (eds) Proceedings of POPL 2008. ACM, New York, pp 273–284. doi:10.1145/1328438.1328472
28. Jongmans SS, Arbab F (2012) Overview of thirty semantic formalisms for Reo. Sci Ann Comput Sci 22(1):201–251. doi:10.7561/SACS.2012.1.201
29. Jongmans SS, Arbab F (2013) Global consensus through local synchronization. In: Canal C, Villari M (eds) Proceedings of FOCLASA 2013, no. 393 in CCIS. Springer, Berlin, pp 174–188. doi:10.1007/978-3-642-45364-9_15
30. Jongmans SS, Arbab F (2013) Global consensus through local synchronization (Technical Report). Tech. Rep. FM-1303, CWI
31. Jongmans SS, Arbab F (2013) Modularizing and specifying protocols among threads. In: Gay S, Kelly P (eds) Proceedings of PLACES 2012, EPTCS, vol 109. CoRR, pp 34–45. doi:10.4204/EPTCS.109.6
32. Jongmans SS, Arbab F (2014) Toward sequentializing overparallelized protocol code. In: Lanese I, Lluch-Lafuente A, Sokolova A, Torres-Vieira H (eds) Proceedings of ICE 2014, EPTCS, vol 166. CoRR, pp 38–44. doi:10.4204/EPTCS.166.5
33. Jongmans SS, Arbab F (2015) Can high throughput atone for high latency in compiler-generated protocol code? In: Dastani M, Sirjani M (eds) Proceedings of FSEN 2015, LNCS. Springer, Berlin (in press)
34. Jongmans SS, Clarke D, Proença J (2012) A procedure for splitting processes and its application to coordination. In: Kokash N, Ravara A (eds) Proceedings of FOCLASA 2012, EPTCS, vol 91. CoRR, pp 79–96. doi:10.4204/EPTCS.91.6
35. Jongmans SS, Santini F, Arbab F (2013) Partially-distributed coordination with Reo (Technical Report). Tech. Rep. FM-1304, CWI
36. Jongmans SS, Santini F, Arbab F (2014) Partially-distributed coordination with Reo. In: Aldinucci M, D'Agostino D, Kilpatrick P (eds) Proceedings of PDP 2014. IEEE, Los Alamitos, pp 697–706. doi:10.1109/PDP.2014.19
37. Jongmans SS, Santini F, Sargolzaei M, Arbab F, Afsarmanesh H (2012) Automatic code generation for the orchestration of web services with Reo. In: de Paoli F, Pimentel E, Zavattaro G (eds) Proceedings of ESOCC 2012, LNCS, vol 7592. Springer, Berlin, pp 1–16. doi:10.1007/978-3-642-33427-6_1
38. Jongmans SS, Santini F, Sargolzaei M, Arbab F, Afsarmanesh H (2014) Orchestrating web services using Reo: from circuits and behaviors to automatically generated code. Serv Oriented Comput Appl 8(4):277–297. doi:10.1007/s11761-013-0147-1
39. Jordan D, Evdemon J (2007) Web services business process execution language version 2.0. Standard ws-bpel-v2.0-OS, OASIS
40. Kokash N, Changizi B, Arbab F (2010) A semantic model for service composition with coordination time delays. In: Dong JS, Zhu H (eds) Proceedings of ICFEM, LNCS, vol 6447. Springer, Berlin, pp 106–121. doi:10.1007/978-3-642-16901-4_9

41. Kokash N, Krause C, de Vink E (2012) Reo+mCRL2: a framework for model-checking dataflow in service compositions. Form Asp Comput 24(2):187–216. doi:10.1007/s00165-011-0191-6
42. Lanese I, Montesi F, Zavattaro G (2013) Amending choreographies. In: Ravara A, Silva J (eds) Proceedings of WWV 2013, EPTCS, vol 123. CoRR, pp 34–48. doi:10.4204/EPTCS.123.5
43. Lynch N (1996) Distributed algorithms. Elsevier, Amsterdam
44. Mamei M, Zambonelli F (2004) Programming pervasive and mobile computing applications with the tota middleware. In: Proceedings of the second IEEE international conference on pervasive computing and communications (PerCom'04), PERCOM '04, p 263. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=977406.978680
45. Mamei M, Zambonelli F (2005) Field-based coordination for pervasive multiagent systems (Springer series on agent technology). Springer, Secaucus
46. Mhl G, Fiege L, Pietzuch P (2010) Distributed event-based systems, 1st edn. Springer, Berlin
47. Milner R (1989) Communication and concurrency. Prentice Hall, Upper Saddle River
48. Mostarda L, Marinovic S, Dulay N (2010) Distributed orchestration of pervasive services. In: Rahayu W, Xhafa F, Denko M (eds) Proceedings of AINA 2010. IEEE, Los Alamitos, pp 166–173. doi:10.1109/AINA.2010.100
49. Murphy AL, Picco GP, Roman GC (2006) Lime: a coordination model and middleware supporting mobility of hosts and agents. ACM Trans Softw Eng Methodol 15(3):279–328. doi:10.1145/1151695.1151698
50. Muth P, Wodtke D, Weissenfels J, Dittrich AK, Weikum G (1998) From centralized workflow specification to distributed workflow execution. J Intell Inf Syst 10(2):159–184. doi:10.1023/A:1008608810770
51. Nanda MG, Chandra S, Sarkar V (2004) Decentralizing execution of composite web services. In: Schmidt D (ed) Proceedings of OOPSLA 2004. ACM, New York, pp 170–187. doi:10.1145/1028976.1028991
52. de Nicola R, Ferrari GL, Pugliese R (1998) KLAIM: a kernel language for agents interaction and mobility. IEEE Trans Softw Eng 24(5):315–330. doi:10.1109/32.685256
53. Web services business process execution language (2007). http://docs.oasis-open.org/wsbpel/2.0/
54. Parr T (2007) Generating structured text with templates and grammars. In: The definitive ANTLR reference: building domain-specific languages. The Pragmatic Bookshelf, pp 208–242
55. Pautasso C (2009) RESTful Web service composition with BPEL for REST. Data Knowl Eng 68(9):851–866. doi:10.1016/j.datak.2009.02.016
56. Pedraza G, Estublier J (2009) Distributed orchestration versus choreography: the FOCAS approach. In: Wang Q, Garousi V, Madachy R, Pfahl D (eds) Proceedings of ICSP 2009, no. 5543 in LNCS. Springer, Berlin, pp 75–86. doi:10.1007/978-3-642-01680-6_9
57. Peltz C (2003) Web services orchestration and choreography. Computer 36(10):46–52. doi:10.1109/MC.2003.1236471
58. Proença J, Clarke D, de Vink E, Arbab F (2011) Decoupled execution of synchronous coordination models via behavioural automata. In: Mousavi MR, Ravara A (eds) Proceedings of FOCLASA 2011, EPTCS, vol 58. CoRR, pp 65–79. doi:10.4204/EPTCS.58.5
59. Proença J, Clarke D, de Vink E, Arbab F (2012) Dreams: a framework for distributed synchronous coordination. In: Viroli M, Castelli G, Marquez JLF (eds) Proceedings of SAC 2012. ACM, New York, pp 1510–1515. doi:10.1145/2245276.2232017
60. Proença J (2011) Synchronous coordination of distributed components. Ph.D. thesis, Leiden University
61. Ren S, Yu Y, Chen N, Marth K, Poirot PE, Shen L (2006) Actors, roles and coordinators – a coordination model for open distributed and embedded systems. In: Proceedings of the 8th international conference on coordination models and languages, COORDINATION'06. Springer, Berlin, pp 247–265. doi:10.1007/11767954_16
62. Ricci A, Omicini A, Viroli M, Gardelli L, Oliva E (2007) Cognitive stigmergy: towards a framework based on agents and artifacts. In: Proceedings of the 3rd international conference on environments for multi-agent systems III, E4MAS'06. Springer, Berlin, pp 124–140. http://dl.acm.org/citation.cfm?id=1759343.1759352
63. Rowstron A, Wood A (1997) BONITA: a set of tuple space primitives for distributed coordination. In: El-Rewini H (ed) Proceedings of HICSS 1997. IEEE, Los Alamitos, pp 379–388. doi:10.1109/HICSS.1997.667285
64. Sirjani M, Jaghoori MM, Baier C, Arbab F (2006) Compositional semantics of an actor-based language using constraint automata. Proceedings of COORDINATION 2006, LNCS, vol 4038. Springer, Berlin, pp 281–297. doi:10.1007/11767954_18
65. Sirjani M, Movaghar A, Shali A, de Boer F (2004) Modeling and verification of reactive systems using Rebeca. Fundam Inform 63:385–410
66. Tretola G, Zimeo E (2006) Workflow fine-grained concurrency with automatic continuation. In: Rosenberg A, Atallah M, Bader D, Gottlieb A, Kale L (eds) Proceedings of IPDPS 2006. IEEE, Los Alamitos, pp 253–260. doi:10.1109/IPDPS.2006.1639510