Centralized coordination vs. partially-distributed coordination with Reo and constraint automata

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

[a] *Open University of the Netherlands, Valkenburgerweg 177, 6419 AT Heerlen, The Netherlands*
[b] *Radboud University Nijmegen, Toernooiveld 212, 6525 EC Nijmegen, The Netherlands*
[c] *Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, The Netherlands*

ARTICLE INFO

ABSTRACT

High-level concurrency notations and abstractions have several well-known software engineering advantages when it comes to programming concurrency protocols among threads. To also explore their complementary performance advantages, in ongoing work, we are developing compilation technology for a high-level coordination language, Reo, based on this language's formal automaton semantics. By now, as shown in our previous work, our tools are capable of generating code that can compete with carefully hand-crafted code, at least for some protocols. An important prerequisite to further advance this promising technology, now, is to gain a better understanding of how the significantly different compilation approaches that we developed so far, which vary in the amount of parallelism in their generated code, compare against each other. For instance, to better and more reliably tune our compilers, we must learn under which circumstances parallel protocol code, with high throughput but also high latency, outperforms sequential protocol code, with low latency but also low throughput.

In this paper, we report on an extensive performance comparison between these approaches for a substantial number of protocols, expressed in Reo. Because we have always formulated our compilation technology in terms of a general kind of communicating automaton (i.e., constraint automata), our findings apply not only to Reo but, in principle, to any language whose semantics can be defined in terms of such automata.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

1.1. Context

In the early 2000s, hardware manufacturers shifted their attention from manufacturing faster—but purely sequential—unicore processors to manufacturing slower—but increasingly parallel—multicore processors. In the wake of this shift, concurrent programming became essential for writing scalable programs on general hardware. Conceptually, concurrent programs consist of (sequential) *processes*, which implement modules of computation, and (concurrency) *protocols*, which implement the rules of interaction that processes must abide by. As programmers have been writing sequential code for

---

(a) Alternator$_2$ (b) Alternator$_3$ (c) EarlyAsyncBarrierMerger$_2$ (d) EarlyAsyncMerger$_2$

(e) EarlyAsyncOutSequencer$_2$ (f) EarlyAsyncReplicator$_2$ (g) LateAsyncMerger$_2$

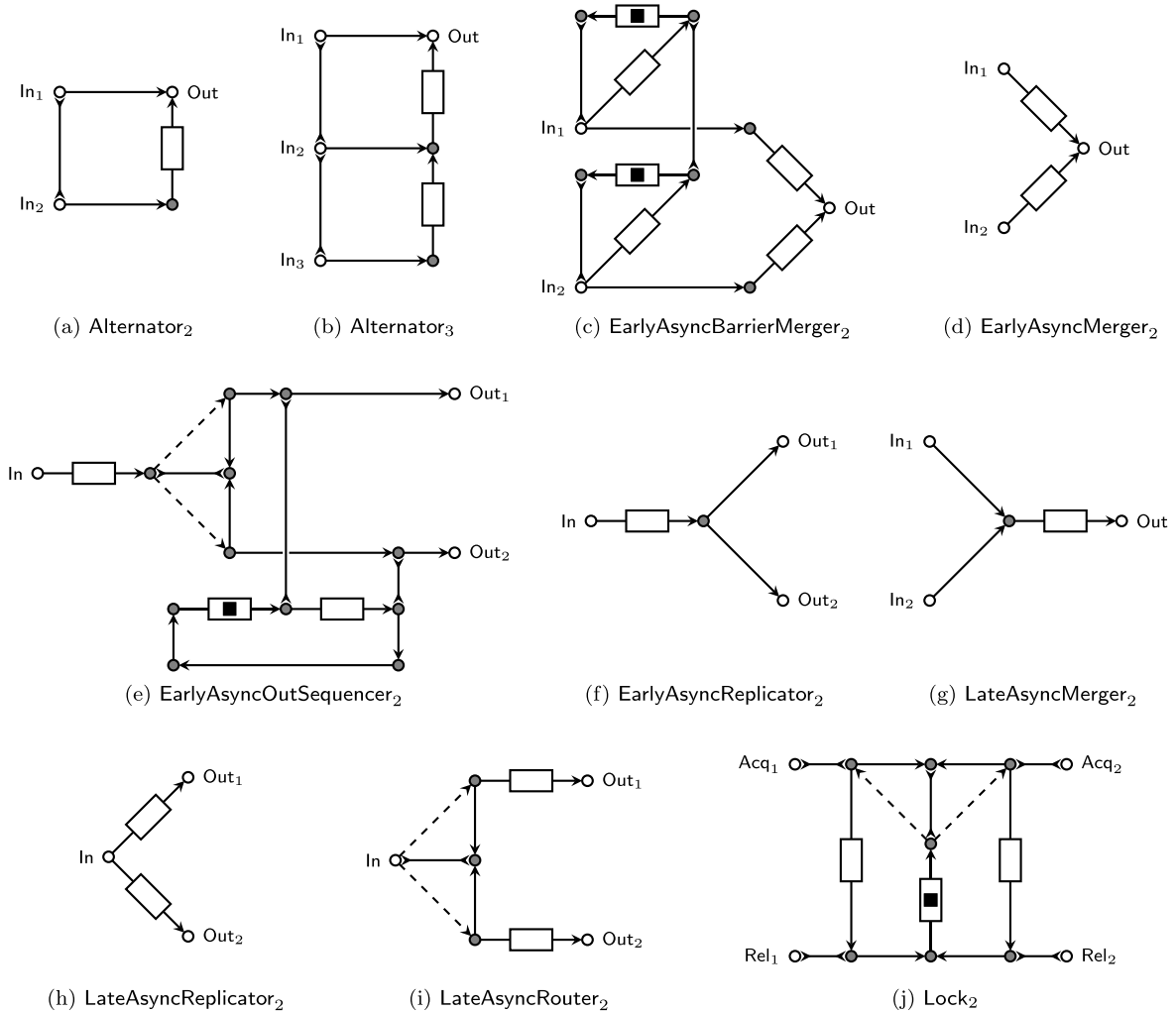(h) LateAsyncReplicator$_2$ (i) LateAsyncRouter$_2$ (j) Lock$_2$

Fig. 1. Example connectors (ordered alphabetically).

decades, programming of processes poses no new fundamental challenges. What *is* new—and notoriously difficult—is programming of protocols.

In ongoing work, we study an approach to concurrent programming based on syntactic separation of processes from protocols. In this approach, programmers write their processes in a *general-purpose language* (GPL), while they write their protocols in a complementary *domain-specific language* (DSL). Paraphrasing the definition of DSLs by Van Deursen et al. [1], a DSL for protocols "is a programming language that offers, through appropriate notations and abstractions, expressive power focused on, and [..] restricted to, [programming protocols]."

Low-level synchronization constructs (e.g., locks or semaphores) make the programming of protocols prone to errors and misbehavior. On the other hand, there are *coordination models* and *languages* [2] with high-level abstractions that let programmers specify the intended protocols of interaction and provide a correct implementation of said protocols. In developing DSLs for protocols, we draw inspiration from the latter. Significant as the software engineering advantages of coordination models and languages may be, however, performance is an important concern, too. A crucial step toward adoption of coordination models and languages for programming of protocols is, therefore, the development of efficient compilers. Such compilers should be capable of generating efficient (in terms of both time and memory) lower-level protocol implementations from higher-level protocol specifications.

Our current work focuses on developing compilation technology for the coordination language Reo [3,4]: a domain-specific language for protocols. Reo facilitates compositional construction of protocol specifications manifested as *connectors*: channel-based mediums through which processes can communicate with each other. Briefly, in Reo, a connector consists of one or more *channels*, through which data items flow, and a number of *nodes*, on which channel ends coincide. To give an impression, Fig. 1 shows a number of example connectors in Reo's graphical syntax; we discuss the meaning of one of these connectors in detail in Section 2 and the others in Appendix A.
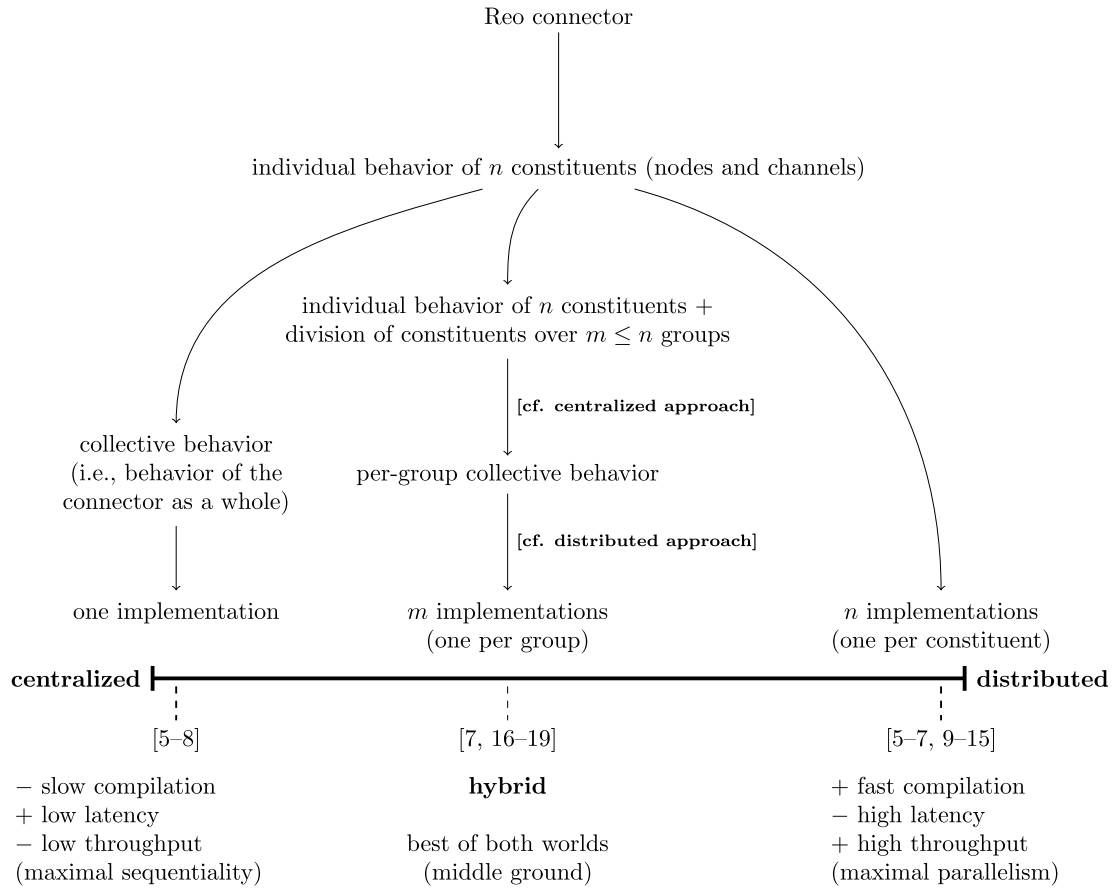
Reo connector

↓

individual behavior of n constituents (nodes and channels)

individual behavior of n constituents +
division of constituents over $m \leq n$ groups

[cf. centralized approach]

collective behavior
(i.e., behavior of the
connector as a whole)

per-group collective behavior

[cf. distributed approach]

one implementation

m implementations
(one per group)

n implementations
(one per constituent)

centralized ├─────────────────────────────────────┤ **distributed**

[5–8]

[7, 16–19]

[5–7, 9–15]

− slow compilation
+ low latency
− low throughput
(maximal sequentiality)

hybrid

best of both worlds
(middle ground)

+ fast compilation
− high latency
+ high throughput
(maximal parallelism)

Fig. 2. Connector compilation spectrum (thick line in the middle), including an overview of the workings of the three main compilation approaches (above the thick line) and their characteristics (below the thick line).
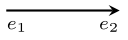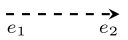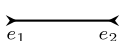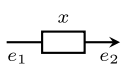
1.2. Problem

Three main approaches for compiling Reo connectors exist. Two of these approaches constitute the ends of the *connector compilation spectrum* in Fig. 2: the further we get to the left end of this spectrum, the more centralized compiler-generated connector implementations become.

- In the *distributed approach* [5–7,9–15], a compiler implements the behavior of each of the n constituents of a connector (i.e., its nodes and its channels) and runs these n implementations in parallel as a distributed system. See also Fig. 2.
  The distributed approach yields maximal parallelism, in the sense that every connector constituent constitutes a distinct unit of parallelism. It has the advantages of fast compilation at build-time and high throughput at run-time. However, this comes at the cost of higher latency at run-time (because of a necessary distributed consensus algorithm). See also Fig. 2.
- In the *centralized approach* [5–8], a compiler computes the behavior of a connector as a whole, implements this behavior, and runs this implementation sequentially as a centralized system. See also Fig. 2.
  Contrasting the distributed approach, the centralized approach yields maximal sequentiality. It has the advantage of low latency at run-time. However, this comes at the cost of slower compilation at build-time and lower throughput at run-time. See also Fig. 2.
- Proença et al. [14,15] observe that a partially-distributed, partially-centralized *hybrid approach* is generally ideal [7, 16–19]. In the hybrid approach, a compiler splits a connector into parts, implements those parts according to the centralized approach, and runs those implementations according to the distributed approach. See also Fig. 2.
  The hybrid approach should strike a perfect middle ground between latency (sequentiality) and throughput (parallelism) at run-time while achieving reasonably fast compilation at build-time. See also Fig. 2.

We started developing a centralized-approach compiler. Subsequently, we gradually moved to a hybrid-approach version. The latter's advantage of fast compilation at build-time constituted our main motivation for this transition. Before this paper, however, we had only little understanding of the implications with respect to run-time performance. Moreover, we

Table 1
Graphical syntax and informal semantics of common channels.

Syntax	Semantics
$e_1 \qquad e_2$	Synchronously [accepts a data item d through its source end e_1 and offers d through its sink end e_2].
$e_1 \qquad e_2$	Synchronously [accepts a data item d through its source end e_1 and nondeterministically [either offers d through its sink end e_2 or loses d]].
$e_1 \qquad e_2$	Synchronously [accepts data items through both its source ends and loses them].
$e_1 \qquad e_2$ (buffer x)	Asynchronously [accepts a data item d through its source end e_1 and stores d in a buffer x], then [offers d through its sink end e_2 and clears x].

recently found a case where hybrid-approach compilation actually took much longer than centralized-approach compilation. This made us realize that we *must* improve our understanding of the differences between the centralized approach and the hybrid approach to advance our compilation technology.

1.3. Contribution

In this paper, we compare centralized-approach compilation and execution with hybrid-approach compilation and execution, focusing on performance in terms of time; we do not consider memory in this paper. Previously studying the theoretical foundations of the hybrid approach [16–19], we have not performed such an experimental evaluation before. For this comparison, we use nine different connector "families" (i.e., connectors parametric in their number of coordinated processes). "Members" of these nine families are shown in Fig. 1. Our comparison reveals previously unknown strengths and weaknesses of the approaches under investigation. These new insights seem imperative for the future development of our compilation technology. Inspired by our experimental results, we also present a general framework to mend one of the weaknesses of the hybrid approach.

Although framed in the context of Reo, our technology works at the more general level of Reo's formal automaton semantics. This formal automaton semantics is based on *constraint automata* [20,21]. Therefore, our findings are of relevance to any high-level model or language whose semantics can be defined in terms of constraint automata. We expect this generality to make our work interesting to a larger audience, beyond the Reo community.

In Section 2, we discuss preliminaries on Reo and its automaton semantics. In Section 3, we present a centralized-approach and a hybrid-approach compiler for Reo. We implemented these compilers from scratch to perform the experiments reported on in this paper. In Section 4, we explain our experimental setup. In Sections 5 and 6, we discuss our experimental results: in Section 5, we discuss results related to the *compilation* of our experimental connectors, while in Section 6, we discuss results related to their *execution*. In Section 7, we present a general framework to improve hybrid-approach compilation, inspired by our findings in Sections 5 and 6. In Section 8, we present related work. Section 9 concludes this paper.

Parts of the material presented in this paper were previously presented at the 6th IPM International Conference on Fundamentals of Software Engineering (FSEN 2015) [22] and at the 7th Interaction and Concurrency Experience (ICE 2014) [23]. The new material in this paper consists of (i) a more self-contained presentation of our compilation technology and its associated programming model, (ii) a more comprehensive analysis of our experimental results, previously scattered over the two aforementioned publications, (iii) a general framework to improve hybrid-approach compilation, and (iv) an overview of related work.

2. Background

2.1. Reo

Reo is a language for compositional construction of concurrency protocols, manifested as connectors. Connectors consist of channels and nodes, organized in a graph-like structure. The rest of this section gives a brief overview of Reo; the standard reference is [3], and a more modern (and gentle) introduction is [4].

Table 1 shows four common channels. Every channel consists of two *ends* and a constraint that relates the timing and the content of the data-flows at those ends. Reo features an open-ended set of channels, which means that programmers can define their own channels with custom semantics. The only rule that user-defined channels must abide by is that every 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), while *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 and allows, for instance, channels with two source ends.

Channel ends coincide on nodes. Contrasting channels, every node behaves in the same way: repeatedly, it nondeterministically selects an available data item out of one of its coincident sink ends and replicates this data item into each of its coincident source ends. A node's nondeterministic selection and its subsequent replication constitute one atomic execution

step (i.e., an atomic data-flow through a connector); nodes cannot temporarily store, generate, or lose data items. A node with only coincident source ends is called a *source node*; one with only coincident sink ends is called a *sink node*; one with both coincident source ends and coincident sink ends is called a *mixed node*. Source and sink nodes constitute the *boundary nodes* of a connector. The boundary nodes of a connector admit I/O operations—`writes` to send and `takes` to receive—from processes; a connector uses its mixed nodes only for internal routing of data items. In figures (e.g., Fig. 1), we distinguish the white, named boundary nodes of a connector from its shaded, anonymous mixed nodes.

Before a connector makes a global execution step, usually instigated by pending I/O operations, its channels and its nodes must have reached *consensus* about their behavior to guarantee mutual consistency of their local execution steps (e.g., a node cannot replicate a data item into a channel with an already full buffer). Afterward, connector-wide data-flow emerges. Note that, as Reo supports both synchronous and asynchronous channels, programmers using Reo can mix both synchronous and asynchronous communication within the same protocol.

Conceptually, using Reo, every process primarily performs sequential computation, locally, using its own logically private memory. Additionally, a process can exchange data items with its environment by performing value-passing blocking input/output operations exclusively on its own local ports. A subsequent binding operation that composes a set of processes and connectors into a full concurrent application, establishes a one-to-one mapping between the set of ports of processes and the set of boundary nodes of connectors in the application: it identifies an output port of a process with a source node of a connector, and an input port of a process with a sink node of a connector. This identification ensures that at run-time, every process has access to a number of boundary nodes (of one or more connectors). A process can interact with other processes *only* through the boundary nodes accessible to it; moreover, the value-passing semantics of the I/O operations available to processes means that logically, processes have no shared memory to directly exchange data with each other. To interact with another process, thus, a process needs to perform a I/O operation on one of its own ports, which maps to a unique boundary node; subsequently, it waits until that operation *completes.*

Although Reo stipulates I/O operations to be blocking, `write` operations can effectively be made partially nonblocking by binding a port of a process to the source end of an asynchronous channel in a connector: so long as the buffer of the asynchronous channel is not full, a `write` will immediately complete, and only once the buffer has become full, a subsequent `write` is blocking. (To support completely nonblocking operations, the original paper on Reo also features asynchronous channels with unbounded buffers [3], but we do not consider those further in this paper.) This use of asynchronous channels also shows that using Reo, it is unnecessary—and actually discouraged—to implement buffers explicitly as (simplistic) processes. On the contrary, buffers are part of protocols and should therefore be part of connectors, where they can be analyzed, verified, reasoned about, and compiled in the right (protocol) context and level of abstraction.

Importantly, because a process performs its I/O operations only on its own local ports, whenever a `write` completes, the process that performed this operation does not know whereto the written data item goes. Similarly, whenever a `take` completes, the process that performed this operation does not know wherefrom the taken data item comes. *Only* connectors decide how data items flow between nodes, thereby coordinating the interaction among processes "from outside" (i.e., "exogenously"). The purpose of Reo is to facilitate construction of connectors, and this is why, from a Reo perspective, we consider processes as black boxes (i.e., it is the perspective opposite to processes' obliviousness toward connectors). Only once we need to reason about a concurrent system in full, we need to open the black boxes and look inside to see how processes exactly behave. The formal model that we use to express behavior of Reo connectors (Sect. 2.3) is sufficiently powerful to also express behavior of processes. As such, it is a suitable formal model in which both connectors and processes can be specified and composed, thereby enabling comprehensive reasoning about concurrent systems in full.

2.2. Example: LateAsyncMerger$_2$

To exemplify Reo, Fig. 3 shows a two-producers-single-consumer program, with producers Alice and Bob, and with consumer Carol. As shown in Fig. 3a, the output port of Alice is identified with source node In_1, that of Bob with source node In_2, and the input port of Carol with sink node Out.

Whenever Alice (Bob) `writes` a data item on her (his) accessible source node In_1 (In_2), this data item flows through channel ends e_1, e_2, and e_5 (e_3, e_4, and e_5) into the buffer (unless this buffer is already filled, in which case the `write` suspends until the buffer becomes empty). Alice (Bob) can immediately continue, possibly before Carol has completed a `take` for the data item in the buffer (i.e., communication between Alice/Bob and Carol transpires asynchronously). Whenever Carol `takes` a data item from her accessible sink node Out, the connector empties the hitherto full buffer. Carol `takes` data items in the order in which Alice/Bob `write` them (i.e., communication between Alice/Bob and Carol transpires undisrupted). Note that the connector among Alice, Bob, and Carol is identical to the LateAsyncMerger$_2$ connector in Fig. 1g (although in Fig. 3a, we also annotated the connector with names for channel ends). The behavior of the other connectors in Fig. 1 is explained in Appendix A.

Fig. 3c shows possible implementations of Alice, Bob, and Carol in Java[1]: method `Producer.main` implements Alice and Bob, while method `Consumer.main` implements Carol. Each of these methods has a formal `SrcNode` parameter

---

[1] Conceptually, there is nothing Java-specific about this example, though: the approach works equally well for other GPLs, and our choice for Java is in that sense arbitrary.
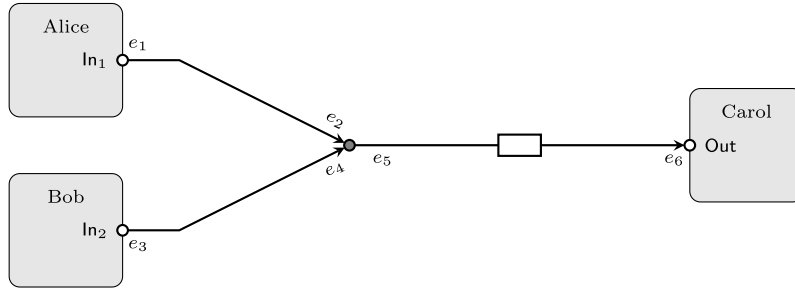
(a) Graphical representation of two producers and a consumer (Alice, Carol, and Bob) and a protocol among them (connector $LateAsyncMerger_2$, Figure 1g)

```
public interface SrcNode {
  public void write(Object obj);
}
```

```
public interface SnkNode {
  public void take(Object obj);
}
```

(b) Java API for nodes

```
public class Producer {
  public static void main(SrcNode src, int id) {
    String dataItem = id + ": Hello, World!";
    while (true)
      src.write(dataItem);
} }
```

```
public class Consumer {
  public static void main(SnkNode snk) {
    while (true) {
      String dataItem = (String) snk.take();
      System.out.println(dataItem);
} } }
```

(c) Hand-written Java code for processes

```
public class Protocol extends Thread {
  private SrcNode In1;
  private SrcNode In2;
  private SnkNode Out;

  public Protocol(
      SrcNode In1, SrcNode In2, SnkNode Out) {

    this.In1 = In1;
    this.In2 = In2;
    this.Out = Out;
  }

  public void run() {
    // Event-driven code to simulate a connector:
    ...
} }
```

```
public class Program {
  public static void main(String[] args) {
    final SrcNode In1 = new SrcNodeImpl();
    final SrcNode In2 = new SrcNodeImpl();
    final SnkNode Out = new SnkNodeImpl();

    // Alice
    (new Thread() { public void run() {
        Producer.main(In1,1); } } ).start();

    // Bob
    (new Thread() { public void run() {
        Producer.main(In2,2); } } ).start();

    // Carol
    (new Thread() { public void run() {
        Consumer.main(Out); } } ).start();

    // Protocol
    (new Protocol(In1,In2,Out)).start();
} }
```

(d) Compiler-generated Java code for the protocol and the program

Fig. 3. Programming example.

(for a source node) or a formal SnkNode parameter (for a sink node), which represents the boundary node accessible to Alice, Bob, or Carol. Fig. 3d shows two additional classes: one for the protocol among Alice, Bob, and Carol, and one for the entire program. The latter class, Program, consists of only a main method. In this method, first, two SrcNodes and a SnkNode are constructed. Subsequently, two method calls to Producer.main (for Alice and Bob) and one method call to Consumer.main (for Carol) are made, each wrapped in a separate Java thread. Finally, a Protocol, which extends Thread, is constructed and started.

The binding operation that composes these independent threads into a concurrent application uses parameter-passing to establish the one-to-one identification of ports with boundary nodes. This works by passing the same SrcNode and SinkNode objects to methods Producer.main/Consumer.main and the constructor of Protocol. As a result, conceptually, the boundary nodes in Fig. 3a become accessible to Alice, Bob, and Carol. The run method in Protocol simulates the $LateAsyncMerger_2$ connector in an event-driven fashion, by monitoring the SrcNodes and SnkNode passed to its constructor; the details of this simulation do not matter in this paper and appear elsewhere [7].

Whereas the code in Fig. 3c is *hand-written* by programmers, the code in Fig. 3d—both classes—is fully *compiler-generated* from a high-level diagram, such as the one in Fig. 3a. This means that programmers do not need to know anything about
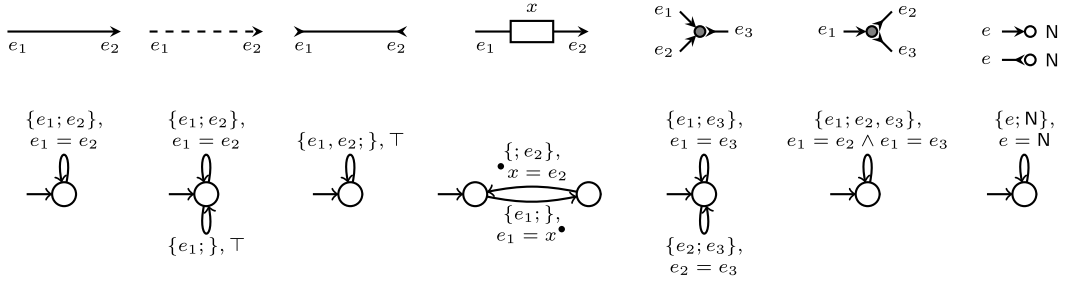
Fig. 4. Constraint automata for the channels in Table 1 (first four from the left), for a mixed node with two incoming and one outgoing channels (fifth from the left), for a mixed node with one incoming and two outgoing channels (sixth from the left), and for two boundary nodes, each with either one incoming or one outgoing channel (seventh from the left). The latter CA is defined not only over the names of its coincident channel ends but also over its own name. (The names of boundary nodes, instead of the names of their coincident channel ends, are used to bind local ports of processes with boundary nodes, thereby providing processes access to these nodes. Therefore, names of boundary nodes must explicitly occur in their CA semantics.).

the concurrency model of Java (e.g., the `Thread` API). Instead, programmers only need to write modules of sequential code and I/O operations on nodes.

The generation of a `Program` class, with its `main` method, is fairly straightforward; the challenging part is compiling connectors into efficient `Protocol` classes. We focus on the latter in the rest of this paper (in terms of time; we do not consider memory in this paper).

2.3. Constraint automata

Our compilers generate code for Reo connectors based on their *constraint automaton* (CA) semantics [20,21]. Constraint automata are a general formalism for modeling concurrent systems, better suited for modeling Reo connectors—and the composition of their nodes and channels in particular—than classical automata or traditional process calculi. For Reo, a CA specifies *when* during execution of a connector *which* data items flow *where* (i.e., through which channel ends). Structurally, every CA consists of finite sets of states and transitions, which model a connector's internal configurations and atomic execution steps. Every transition has a label that consists of two elements: (i) a set with the names of those channel ends that have synchronous data-flow, called a *synchronization constraint*, and (ii) a logical formula that specifies which particular data items may flow through which of those ends, called a *data constraint*. Examples of such formulas include $e_1 = e_2$ (meaning: the same data item flows through channel ends e_1 and e_2), $e = x^\bullet$ (meaning: the same data item flows through channel end e and *into* buffer x), $^\bullet x = e$ (meaning: the same data item flows *out of* buffer x and through channel end e) and $\top$ (meaning: any data item may flow through any channel end or into/out of any buffer). Although $=$ is a theoretically symmetric operator, as a notational convention, we usually put source ends on the left-hand side and sink ends on the right-hand side. Let $\mathbb{DC}$ denote the set of all data constraints.

Definition 1. A constraint automaton is a tuple $(Q, E^{\mathrm{src}}, E^{\mathrm{snk}}, X, \longrightarrow, q_0)$, where:

- Q denotes a set of states
- $E^{\mathrm{src}}, E^{\mathrm{snk}}$ denote sets of source ends and sink ends such that $E^{\mathrm{src}} \cap E^{\mathrm{snk}} = \emptyset$
- X denotes a set of buffers
- $\longrightarrow \subseteq Q \times 2^{E^{\mathrm{src}} \cup E^{\mathrm{snk}}} \times \mathbb{DC} \times Q$ denotes a transition relation
- q_0 denotes an initial state

$\mathbb{A}$UTOM denotes the set of all constraint automata. □

We call a CA $\boldsymbol{a}$ *asynchronous*, denoted as $\xrightarrow{1}(\boldsymbol{a})$, whenever each of its transitions has a singleton synchronization constraint; an asynchronous CA never synchronizes multiple of its ends. Note that, as stated before, CAs can also be used to express the behavior of processes, in terms of when they perform I/O operations on which ports (but abstracting away their actual internal computation).

The semantics of a constraint automaton $(Q, E^{\mathrm{src}}, E^{\mathrm{snk}}, X, \longrightarrow, q_0)$ is defined in terms of accepted words and languages. Informally, every word is an infinite sequence $\delta_1, \delta_2, \ldots$, where every δ_i is a (partial) function from (a subset of) $E^{\mathrm{src}} \cup E^{\mathrm{snk}}$ to a set of data items; intuitively, every δ_i represents one atomic data-flow from source ends to sink ends through a connector (i.e., an execution step), where data item $\delta_i(e)$ flows through end e. The language of a constraint automaton is the set of all words on which it can perform an infinite run (every δ_i causes a transition to fire); formal details appear elsewhere [20, 21]. Notational differences aside, the main difference between constraint automata and classical (non)deterministic finite automata is that alphabet symbols (δ_i-s) are represented symbolically (as data constraints) instead of explicitly; this makes it possible to handle infinite data domains with finite automata.

We associate every node and every channel of a connector with a CA for its local behavior. Fig. 4 shows examples of such CAS, where in synchronization constraints, we separate source ends from sink ends by a semicolon. Note that the fourth CA from the right is asynchronous. A *product operator* on CAS subsequently models composition of nodes and channels into arbitrarily complex connectors: to obtain the "big" CA for a whole connector, one can incrementally compute the product of the "small" CAS for its constituent nodes and channels. Let $dc_1 \wedge dc_2$ denote the conjunction of data constraints dc_1 and dc_2, let $\exists e.dc$ denote existential quantification of e in data constraint dc, and let $\exists E.dc$ extend such existential quantification from single elements to sets. Also, let $E_1 \bigtriangleup E_2$ denote the symmetric difference of sets E_1 and E_2.

Definition 2. $\otimes : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ denotes the partial function defined by the following equation:

$$
\begin{pmatrix} Q_1, \\ E_1^{\text{src}}, \\ E_1^{\text{snk}}, \\ X_1, \\ \longrightarrow_1 \\ q_{0,1} \end{pmatrix} \otimes \begin{pmatrix} Q_2, \\ E_2^{\text{src}}, \\ E_2^{\text{snk}}, \\ X_2, \\ \longrightarrow_2 \\ q_{0,2} \end{pmatrix} = \begin{pmatrix} Q_1 \times Q_2 \\ (E_1^{\text{src}} \cup E_2^{\text{src}}) \setminus (E_1^{\text{snk}} \cup E_2^{\text{snk}}) \\ (E_1^{\text{snk}} \cup E_2^{\text{snk}}) \setminus (E_1^{\text{src}} \cup E_2^{\text{src}}) \\ X_1 \cup X_2 \\ \longrightarrow_\otimes \\ (q_{0,1}, q_{0,2}) \end{pmatrix} \text{ if } \begin{bmatrix} (E_1^{\text{src}} \cap E_2^{\text{src}}) = (E_1^{\text{snk}} \cap E_2^{\text{snk}}) = \emptyset \\ \textbf{and } X_1 \cap X_2 = \emptyset \end{bmatrix}
$$

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

$$
\frac{q_1 \xrightarrow{E_1,dc_1} q_1' \text{ and } q_2 \xrightarrow{E_2,dc_2} q_2'}{\textbf{and } E_1^{\text{all}} \cap E_2 = E_2^{\text{all}} \cap E_1} \\ \frac{}{(q_1,q_2) \xrightarrow{E_1 \bigtriangleup E_2, \exists E_1 \cap E_2.(dc_1 \wedge dc_2)}_\otimes (q_1',q_2')}
\qquad
\frac{q_1 \xrightarrow{E_1,dc_1} q_1' \text{ and } q_2 \in Q_2}{\textbf{and } E_2^{\text{all}} \cap E_1 = \emptyset} \\ \frac{}{(q_1,q_2) \xrightarrow{E_1,dc_1 \wedge [X_2]}_\otimes (q_1',q_2)}
\qquad
\frac{q_2 \xrightarrow{E_2,dc_2} q_2' \text{ and } q_1 \in Q_1}{\textbf{and } E_1^{\text{all}} \cap E_2 = \emptyset} \\ \frac{}{(q_1,q_2) \xrightarrow{E_2,dc_2 \wedge [X_1]}_\otimes (q_1,q_2')}
$$

where $E_1^{\text{all}} = E_1^{\text{src}} \cup E_1^{\text{snk}}$ and $E_2^{\text{all}} = E_2^{\text{src}} \cup E_2^{\text{snk}}$. □

The meaning of predicate $[X]$ in the latter two rules does not matter in this paper (but omitting it would yield a technically incorrect definition) and is explained elsewhere [7].

In words, the product operator consumes CAS $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ as input and produces a CA $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ as output if two conditions hold. The first condition states that the ends shared between $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ must have compatible directions; the second condition states that $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ may not share buffers, essentially to prevent data races among CAS. If these conditions hold, we take the cartesian product of the sets of states, the "direction-sensitive" difference of the sets of ends, the union of the sets of buffers, a new transition relation, and the pair of the initial states. For the computation of a new transition relation, we use three rules. The first rule states that a transition in $\boldsymbol{a}_1$ involving a shared end can fire iff a transition in $\boldsymbol{a}_2$ involving that same shared end also fires. In other words, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ must *agree* on instantaneously firing transitions involving exactly the same shared ends (the conjunct on the second line in the premise). The compound transition, then, is labeled with a synchronization constraint consisting of all unshared ends of the constituent transitions (i.e., the shared ends are abstracted away and become unobservable) and with a data constraint consisting of an existential quantification for the shared ends over the two data constraints of the constituent transitions. The second rule states that a transition in $\boldsymbol{a}_1$ involving no shared ends can fire at any time. The third rule is symmetric to the second one. Figs. 5 and 6 show an example of computing the product expression for the connector in Fig. 3a. Note that in Fig. 6, we can eliminate the $\exists$ quantifiers by iteratively exploiting equivalence $\exists b.(a = b \wedge b = c) \equiv a = c$. For $\exists$ quantifiers that are added to data constraints by the product operator, our compilers perform this optimization automatically [16]; this can always be done for the data constraints that we consider in this paper. Whether or not quantifiers can be eliminated in arbitrary data constraints in general depends on the particular instantiation of $\mathbb{DC}$.

Note that the first rule in Definition 2 also applies to pairwise *independent transitions* in $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, which do not involve any shared ends. As such, because of this first rule, the product operator supports *true concurrency* among pairwise independent transitions.

2.4. Remark about generality

Our compilers operate fully at the level of Reo's CA semantics. Our focus on Reo so far in this paper is therefore misleading: we use Reo's graphical notations and channel-based abstractions just as a—not *the*—programmer-friendly syntax for exposing CA-based protocol programming. Different syntax alternatives for CAS may work equally well or yield perhaps even more user-friendly languages. For instance, we know how to translate UML sequence/activity diagrams and BPMN to CAS [24–26]. Algebras of Bliudze and Sifakis [27,28], originally developed for BIP [29], also have a straightforward interpretation in terms of CAS [30], thereby offering an interesting alternative possible syntax. Due to their generality, CAS can thus serve as an intermediate format for compiling specifications in many different languages and models of concurrency, by reusing the core of our compilers. This makes the development of our compilation technology relevant beyond Reo.

In this paper, for simplicity, we consider only the four Reo channels in Table 1. In principle, however, Reo allows users to define their own channels with custom semantics. For instance, another common channel in the Reo literature is the *filter*
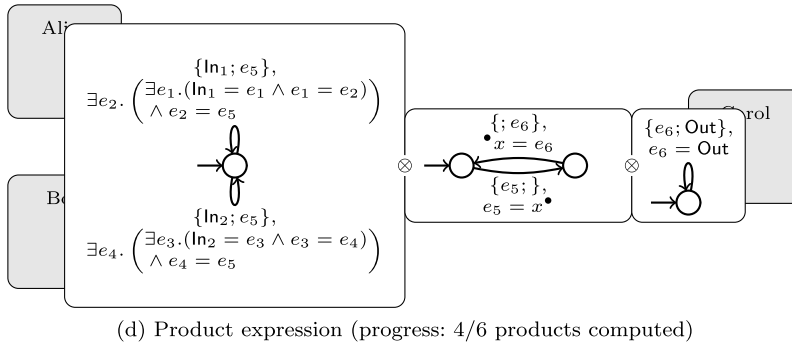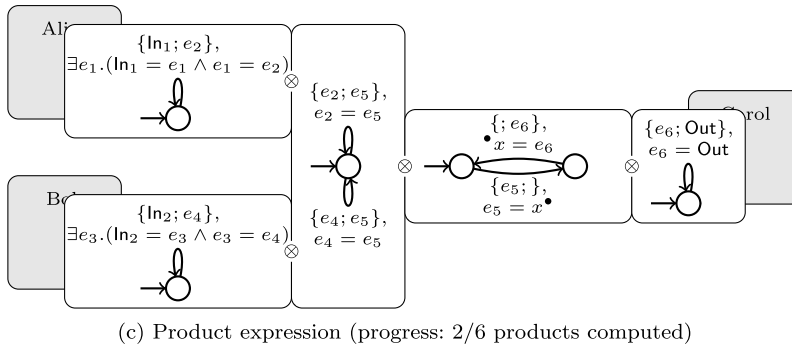
(a) Connector (cf. Figure 3a)

(b) Product expression (progress: 0/6 products computed)

(c) Product expression (progress: 2/6 products computed)

(d) Product expression (progress: 4/6 products computed)
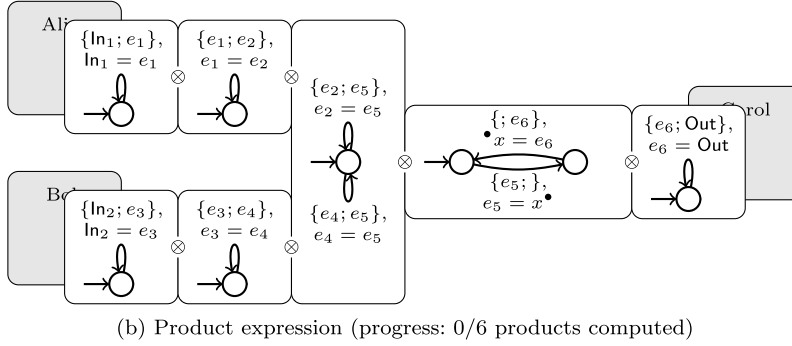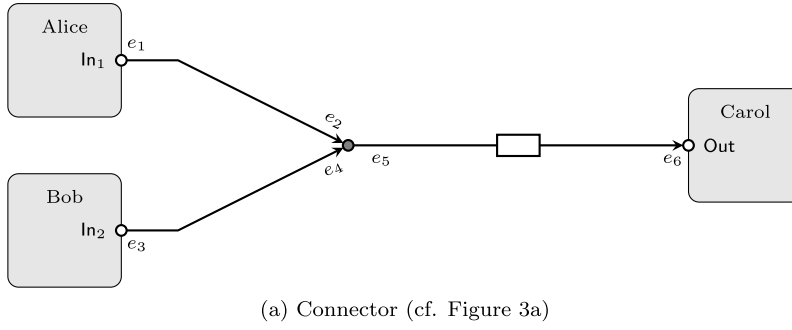
Fig. 5. Constraint automata product example (I).

channel [3]. This channel is similar to the synchronous channel in Table 1 (in the top row), except that it loses data items that do not satisfy a predefined filter predicate. Arbab et al. established (with a constructive proof) that the four channels in Table 1 and the filter channel are *complete* under the CA semantics [31]: for every CA, there exists a corresponding Reo connector consisting of these five channels. A corollary of this result is that these five channels are as expressive as the BIP connector algebra of Bliudze and Sifakis [27]; this is further detailed by Dokter et al. [30].

$$dc_1 = \exists e_5.(\exists e_2.(\exists e_1.(\mathsf{In}_1 = e_1 \wedge e_1 = e_2) \wedge e_2 = e_5) \wedge e_5 = x^\bullet) \equiv \mathsf{In}_1 = x^\bullet$$
$$dc_2 = \exists e_5.(\exists e_4.(\exists e_3.(\mathsf{In}_2 = e_3 \wedge e_3 = e_4) \wedge e_4 = e_5) \wedge e_5 = x^\bullet) \equiv \mathsf{In}_2 = x^\bullet$$
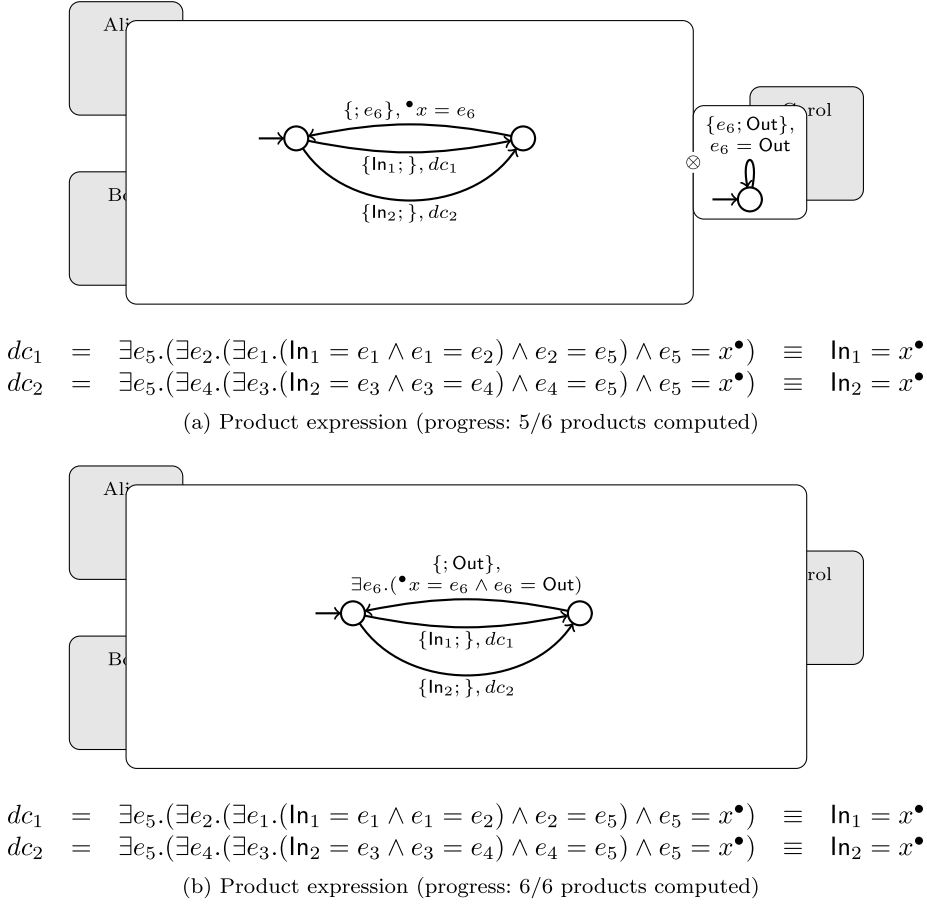
(a) Product expression (progress: 5/6 products computed)

$$dc_1 = \exists e_5.(\exists e_2.(\exists e_1.(\mathsf{In}_1 = e_1 \wedge e_1 = e_2) \wedge e_2 = e_5) \wedge e_5 = x^\bullet) \equiv \mathsf{In}_1 = x^\bullet$$
$$dc_2 = \exists e_5.(\exists e_4.(\exists e_3.(\mathsf{In}_2 = e_3 \wedge e_3 = e_4) \wedge e_4 = e_5) \wedge e_5 = x^\bullet) \equiv \mathsf{In}_2 = x^\bullet$$

(b) Product expression (progress: 6/6 products computed)

Fig. 6. Constraint automata product example (II).

3. Compilers

For our performance comparison, based on earlier implementations [16,19], we developed two Reo/CA-to-Java compilers as mentioned already in Section 1: a centralized-approach one, henceforth referred to as $\mathtt{Compiler_{centr}}$, and a hybrid-approach one, henceforth referred to as $\mathtt{Compiler_{hybr}}$. Both compilers generate shared memory Java code. On input of a connector, $\mathtt{Compiler_{centr}}$ (i) first finds a small CA for every channel and every node that this connector consists of, (ii) then forms the product of all those CAs to get a big CA for the whole connector, and (iii) finally generates one piece of sequential code for that big CA. The size of this piece of code is linear in the number of states and transitions of the big CA. At run-time, this piece of code logically has its own Java thread. Essentially, the construction of a big CA in this way corresponds to parallel expansion in process algebra [32]. $\mathtt{Compiler_{hybr}}$ also first finds a set of small CAs, but in contrast to $\mathtt{Compiler_{centr}}$, it does not form their product to get a big CA. Instead, it computes an m-size *partition* of this set. By doing so, $\mathtt{Compiler_{hybr}}$ effectively splits a connector into a number of "regions" (i.e., connected subconnectors), each of which has a corresponding subset in the partition. After computing a partition, $\mathtt{Compiler_{hybr}}$ forms products on a per-region basis, which results in m "medium" CAs, and generates a piece of sequential code for each of them. As in the centralized approach, the sizes of these pieces of code are linear in the number of states and transitions of their respective medium CAs. At run-time, every such piece of code logically has its own Java thread. These threads use shared memory (plus concurrency protection) to synchronize their actions whenever necessary, using a consensus algorithm.

Fig. 7 pseudomathematically shows centralized-approach compilation and hybrid-approach compilation; for completeness, we also show distributed-approach compilation (essentially, a special case of $\mathtt{Compiler_{hybr}}$ where $n = m$), but we do not consider this kind of compilation further in this paper. Every b_i in b_1 (x) $\cdots$ (x) b_m represents a piece of code generated for a medium CA, while the (x) symbols between them their necessary consensus algorithm.

$\mathtt{Compiler_{hybr}}$'s partitioning algorithm—the crucial element of hybrid-approach compilation—iterates over the set of small CAs and incrementally extends its computed partition (starting from an empty one) [16,19]. For every small CA $\boldsymbol{a}$, the algorithm decides either to add {$\boldsymbol{a}$} to the partition (as a new singleton subset) or to add $\boldsymbol{a}$ to one or more existing parts. (In the latter case, the algorithm subsequently merges all extended subsets into one new subset.) We formulated the condition based on which the algorithm makes this decision generally, in terms of CAs and their transitions [16,19].
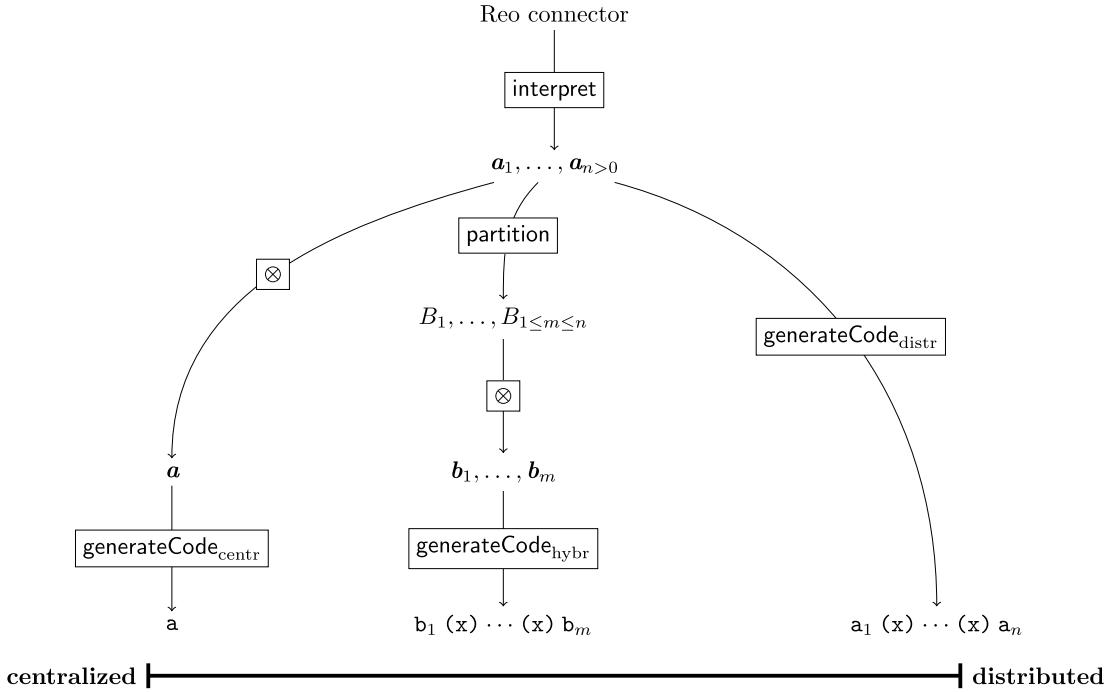
Reo connector

interpret

$a_1, \ldots, a_{n>0}$

partition

$B_1, \ldots, B_{1 \leq m \leq n}$

$\otimes$

$\text{generateCode}_{\text{distr}}$

a

$b_1, \ldots, b_m$

$\text{generateCode}_{\text{centr}}$ $\text{generateCode}_{\text{hybr}}$

a   b$_1$ (x) $\cdots$ (x) b$_m$ a$_1$ (x) $\cdots$ (x) a$_n$

centralized |—————————————————————————————| distributed

Fig. 7. Connector compilation approaches relative to the connector compilation spectrum in Fig. 2.

In the context of Reo, however, this partitioning algorithm coincides with the identification of *synchronous/asynchronous regions* of a connector [15,17] (each of which gets a corresponding subset in the partition). The asynchronous regions of a connector are its smallest connected subconnectors that have asynchronous data-flow (e.g., the fourth channel in Table 1). By removing the asynchronous regions from a connector, its pairwise disconnected synchronous regions remain: largest connected subconnectors with synchronous data-flow. Intuitively, asynchronous regions decouple synchronous regions. Such decoupling enables synchronous regions to run independently of each other: communication between synchronous regions always proceeds in an asynchronous fashion, through a shared asynchronous region. (A connector without asynchronous regions consists of one comprehensive synchronous region. For such connectors, Compiler$_{\text{hybr}}$ reduces to Compiler$_{\text{centr}}$. Conversely, a connector consisting of only asynchronous channels is compiled into a "maximally parallel" implementation, where every channel runs in its own thread. For such connectors, thus, hybrid-approach compilation effectively reduces to distributed-approach compilation, as explained in Sect. 1.2.)

For instance, Fig. 5d shows the medium CAs that result from applying the previously explained partitioning algorithm to the LateAsyncMerger$_2$ connector in Figs. 1g and 3a. This connector consists of two synchronous regions and one asynchronous region between them. The middle CA represents the asynchronous channel in the middle (i.e., one asynchronous region). The leftmost CA represents the synchronous region left of the asynchronous channel (i.e., three nodes, two channels). At run-time, the Java thread for this CA repeatedly makes a choice between its two inputs and passes the data item from the chosen input into the asynchronous channel (i.e., into buffer x). The rightmost CA represents the synchronous region right of the asynchronous channel (i.e., only one node). At run-time, the Java thread for this CA repeatedly passes a data item from the asynchronous channel (i.e., from buffer x) to its output. As a side note, Fig. 6b shows the big CA computed for LateAsyncMerger$_2$ in the centralized approach; Fig. 5b shows the small CAs for LateAsyncMerger$_2$ in the distributed approach.

Notably, a connector represents the logic behind—not the architecture of—the data-flow in a protocol. For instance, even though Lock$_2$ in Fig. 1j, which represents a classical lock, consists of a mix of synchronous, asynchronous, and lossy channels, its compiler-generated code uses neither physical hardware channels nor virtual software channels to realize its desired behavior.

4. Experimental setup

To study under which circumstances code generated by Compiler$_{\text{hybr}}$ outperforms code generated by Compiler$_{\text{centr}}$, we performed a number of experiments. In every experiment, we compared the performance (in terms of time; we do not consider memory in this paper) of centralized and hybrid implementations of a k-parametric connector family, for $k \in \{2, 4, 6, 8, 10, 12, 14, 16, 32, 48, 64\}$. Fig. 1 shows the $k = 2$ members of the nine connector families that we investigated. (One can extend these $k = 2$ members to their $k > 2$ versions in a similar way as how we extended Fig. 1a to

Fig. 1b.) We selected these families because each of them exhibits a different behavior in terms of (a)synchrony, exclusion, nondeterminism, polarity, sequentiality, and parallelism, thereby collectively providing a balanced comparison. In total, thus, we investigated 99 different connectors and twice as many Java implementations. We ran every implementation nine times on a machine with 24 cores (two Intel E5-2690V3 processors with twelve physical cores statically at 2.6 GHz in two sockets, hyperthreading disabled) and averaged our measurements. In every run, we warmed up the JVM for thirty seconds before starting to measure the number of "rounds" that an implementation could finish in the subsequent four minutes. What constitutes one round differs per connector; see below.

Primarily, we wanted to study and measure the overhead of the consensus algorithm among the Java threads in the hybrid implementations (which increases their latency) relative to those implementations' increased parallelism (which increases their throughput). To focus our measurements on only that particular aspect, we needed to eliminate as much as possible all other, orthogonal sources of computation inside compiler-generated code. For this reason, even though our compilers fully support data constraints, we configured our compilers to ignore data constraints during compilation of our experimental connectors (i.e., by replacing every data constraint with ⊤, essentially). As a result, no data processing occurred at run-time during our experiments (i.e., ⊤ is trivially true and requires no run-time resources), which would have constituted a substantial source of sequential computation. Had we enabled data processing, its irrelevant—at least to this comparison—overhead would have polluted our measurements.

For convenience, we divided the connector families under study (Fig. 1) over two categories, except Lock: k-producer–single-consumer and singleproducer-k-consumer. Each of these categories consists of four families. The k-producer-single-consumer category contains LateAsyncMerger (Fig. 1g), EarlyAsyncMerger (Fig. 1d), EarlyAsyncBarrierMerger (Fig. 1c), and Alternator (Figs. 1a and 1b); the single-producer–k-consumer category contains LateAsyncReplicator (Fig. 1h), EarlyAsyncReplicator (Fig. 1f), LateAsyncRouter (Fig. 1i), and EarlyAsyncOutSequencer (Fig. 1e). Appendix A explains the behavior of members of these connector families.

5. Experimental results: compilation

5.1. Measurements

We used Compiler$_{hybr}$ and Compiler$_{centr}$ to compile the connector families in Fig. 1 for the aforementioned values of k with a transition limit of 8096 and a timeout after five minutes. We imposed a transition limit, because the Java compiler cannot conveniently handle Java code generated for CAs with so many transitions;[2] we imposed a compilation timeout, because waiting for longer than five minutes to compile a single connector in practice seems unacceptable. Fig. 8 shows the measured compilation times; per-family figures (with the same curves as in Fig. 8) appear in Appendix B.

For most connector families, Compiler$_{hybr}$ required substantially less time than Compiler$_{centr}$. In fact, for six of our nine connector families, Compiler$_{centr}$ failed to run to completion beyond certain (relatively low) values of k, as witnessed also by their very steep curves in Fig. 8:

- For EarlyAsyncMerger$_{k>7}$, LateAsyncReplicator$_{k>8}$ and LateAsyncRouter$_{k>7}$, the transition number of their "big" CAs exceeded the limit (e.g., EarlyAsyncMerger$_8$ has 23801 transitions, LateAsyncReplicator$_9$ has 19172 transitions, and LateAsyncRouter$_8$ has 23801 transitions) or the compiler timed out.
- For EarlyAsyncBarrierMerger$_{k>4}$, EarlyAsyncOutSequencer$_{k>14}$, and Lock$_{k>12}$, the compiler timed out.

In contrast, Compiler$_{hybr}$ had no problems compiling these connector families for all values of k under investigation. For LateAsyncMerger$_k$ and EarlyAsyncReplicator$_k$, our two compilers required a comparable amount of time for all values of k under investigation. Finally, only for Alternator$_k$, Compiler$_{hybr}$ required substantially more time than Compiler$_{centr}$ did. In this case, Compiler$_{hybr}$ timed out for $k > 12$, while Compiler$_{centr}$ had no problems.

5.2. Discussion

In Section 1, we stated that hybrid-approach compilers have the advantage of "fast compilation at build-time" compared to centralized-approach compilers. The idea behind this statement is that the computation of a big CA in the centralized approach requires many resources, especially whenever this big CA has a size exponential in k. In contrast, hybrid-approach compilers generally compute multiple medium CAs instead of a single big CA; typically, the medium CAs computed for (the regions of) a connector are much smaller than that connector's big CA, because each of those medium CAs is computed out of fewer small CAs than that big CA. As a result, in cases of exponential growth, medium CAs typically have a much smaller exponent than their corresponding big CA. A superficial look at our measurements in Fig. 8 seems to confirm this intuition: compilation time grows exponentially as k increases for each of the six connector families for which Compiler$_{centr}$ eventually failed. Beyond this superficial look, however, there are peculiarities that need clarification.

---

[2] Recall from Section 3 that the size of generated code is linear in the number of states and transitions of big/medium CAs. Java code generated for a CA with excessively many transitions, thus, can be too large for the Java compiler to process within reasonable resource bounds.
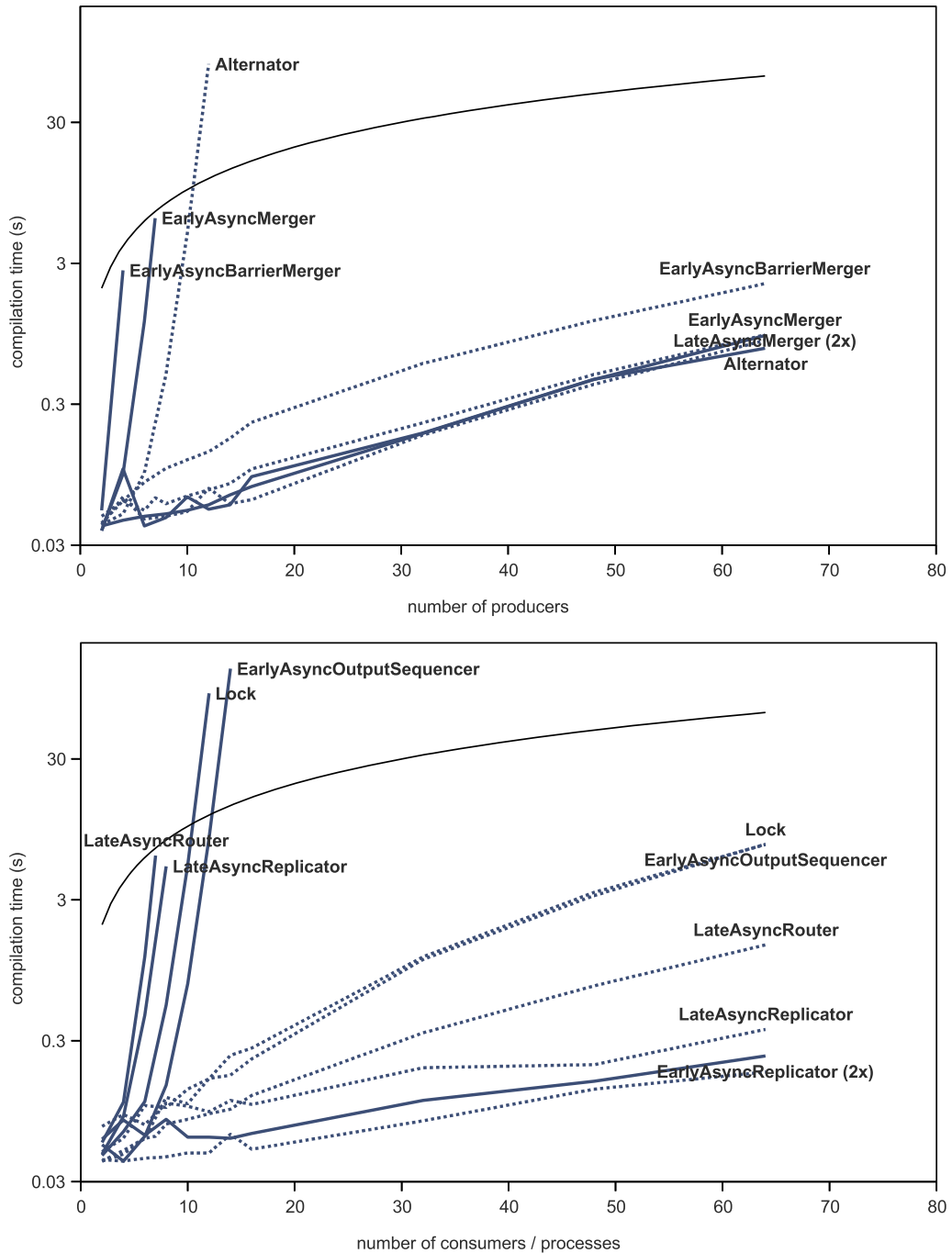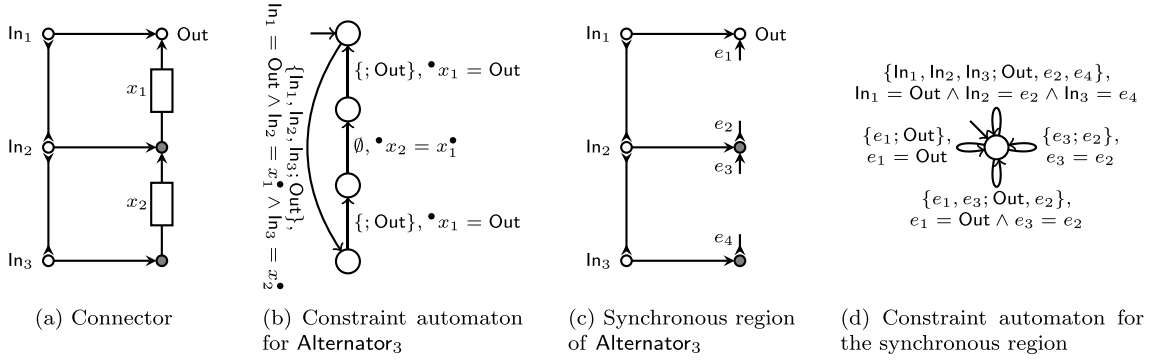
Fig. 8. Compilation times (solid/dotted lines for centralized/hybrid; thinner black curves for proportional growth $x = y$).

- A first peculiarity relates to the measurements for Alternator, which $Compiler_{hybr}$—instead of $Compiler_{centr}$—eventually fails on. To understand these measurements, consider Fig. 9. This figure shows $Alternator_3$, its CA, its single synchronous region, and the CA for this region (including names for four of its channel ends).

  The first transition in Fig. 9b, from the top state to the bottom state, models an execution step of $Alternator_3$ where data items synchronously flow from source node In_1 to sink node Out, from source node In_2 into buffer x_1, and from source node In_3 into buffer x_2. The second transition, then, models an execution step of $Alternator_3$ where a data item flows out of buffer x_1 to Out. The third transition—an unobservable one—models an execution step where a data item flows out of buffer x_2 into buffer x_1. And the fourth transition is the same as the second transition. These transitions correspond to the behavior of $Alternator_3$ as described in terms of completions of writes/takes in Appendix A.

(a) Connector

(b) Constraint automaton for Alternator$_3$

(c) Synchronous region of Alternator$_3$

(d) Constraint automaton for the synchronous region

Fig. 9. Alternator$_3$.

Next, consider Fig. 9d. The "north" transition in this figure corresponds to the first transition in Fig. 9b; the "west" transition corresponds to the second and the fourth transitions in Fig. 9b; the "east" transition corresponds to the third transition in Fig. 9b. Finally, the "south" transition models an execution step of the synchronous region where the execution steps modeled by the west and east transitions *incidentally* occur simultaneously, by true concurrency. An interesting observation about this south transition, however, is that it is, in fact, *permanently disabled* in the environment within which the synchronous region runs (i.e., the two asynchronous regions, each of which consists of a single asynchronous channel). To see this, derive from Table 1 that buffers of asynchronous channels cannot become empty and full again in the same execution step, which is required for the south transition to fire (i.e., buffer x_1 is both emptied and filled in the south transition). Only by computing the product of the CA for the synchronous region of Alternator$_3$ with CAs for its asynchronous regions, the south transition becomes formally recognizable as permanently disabled and is subsequently filtered out (i.e., the south transition violates the premise of every rule in Definition 2 of $\otimes$). This is why Fig. 9b does not have a corresponding transition.

Now, imagine Alternator$_k$ (as in Appendix A). This connector has $k-1$ buffers. Consequently, the medium CA for its single synchronous region has $k-2$ unobservable transitions (among others), each of which models an execution step where a data item flows out of one buffer into the buffer directly above it. Because any subset of those transitions may fire simultaneously, by true concurrency, the medium CA has roughly 2^{k-2} transitions. For instance, the medium CA for Alternator$_{512}$ has over 10^{153} transitions—approximately 10^{73} times the estimated number of hydrogen atoms in the observable universe—meaning that merely representing this CA in memory is already problematic, let alone compositionally computing it. Of course, the bulk of these transitions become permanently disabled in the environment of the asynchronous buffers that this CA connects to; however, `Compiler`$_{hybr}$ does not know this fact a priori, to filter them out. This *transition relation explosion*, then, is exactly why `Compiler`$_{hybr}$ requires exponentially many more resources as k increases, as we observed.

Essentially, thus, a hybrid-approach compiler does not have the required information to treat every mixed node of a connector, which a connector uses only for internal routing of data items, as *truly* internal: the channel ends coincident on mixed nodes that mark the *boundaries* between a connector's regions are not abstracted away in the corresponding medium CAs. In fact, from a practical perspective, they cannot be: at run-time, threads for medium CAs for neighboring regions synchronize their actions through their shared channel ends, which makes it essential to explicitly represent them in compiler-generated code. From a more theoretical perspective, abstraction of channel ends *generally* occurs only whenever those ends are shared in a product of CAs, but because the *particular* channel ends under discussion mark the boundaries between regions, and because hybrid-approach compilers compute products only on a per-region basis, these particular channel ends are never shared in any product computed by a hybrid-approach compiler. Hence, those channel ends never get abstracted away, and many conceptually unobservable transitions may remain, potentially to the extent that these transitions cause transition relation explosion in medium CAs (as with Alternator). In Section 7, we propose a new optimization technique to resolve this issue.

- The second peculiarity concerns centralized-approach compilation. First, by analyzing the big CAs of the k-parametric connector families EarlyAsyncBarrierMerger, EarlyAsyncMerger, LateAsyncReplicator, and LateAsyncRouter, we found that those CAs grow *exponentially* as k increases (due to the many ways in which their k independent transitions can synchronously fire, by true concurrency). This explains why `Compiler`$_{centr}$ requires exponentially more time as k increases to compile members of those families, as shown in Fig. 8. Now, it seems not unreasonable to assume also the inverse: for k-parametric connector families whose big CAs grow only *linearly* in k, `Compiler`$_{centr}$ should scale fine. Alternator, EarlyAsyncReplicator, and LateAsyncMerger, which satisfy this premise, seem to validate this assumption. Indeed, Fig. 8 shows that `Compiler`$_{centr}$ has no problems with compiling members of those families. (The big CAs of the EarlyAsyncReplicator family even have a constant number of transitions.) However, this still leaves us with two families whose compilation behavior we have not yet accounted for: EarlyAsyncOutSequencer and Lock. Although the big

CAS of both these k-parametric families grow only linearly in k, Fig. 8 shows that $\texttt{Compiler}_{\text{centr}}$ nevertheless requires exponentially more time as k increases.

It turns out that even if big CAS grow only linearly in k, the "intermediate products" encountered during their computation may "temporarily" grow exponentially. For instance, if we have three CAS $\boldsymbol{a}_1$, $\boldsymbol{a}_2$ and $\boldsymbol{a}_3$, the intermediate product of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ may grow exponentially in k, while the full product of $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, and $\boldsymbol{a}_3$ grows only linearly. This is easiest to explain for EarlyAsyncOutSequencer$_k$ (cf. Fig. 1e), in terms of its number of states. EarlyAsyncOutSequencer$_k$ consists of a subconnector that, in turn, consists of a cycle of k buffered channels (of capacity 1). The first buffered channel initially contains a dummy data item ■ (i.e., its actual value does not matter); the other buffered channels initially contain nothing. As in the literature [3,4], we call this subconnector Sequencer$_k$. Because no new data items can flow into Sequencer$_k$, only ■ cycles through the buffers—ad infinitum—such that only one buffer holds a data item at any time. Consequently, the CA for Sequencer$_k$ has only k states, each of which represents the presence of ■ in a different one of the k buffers. However, when $\texttt{Compiler}_{\text{centr}}$ compositionally computes this CA out of a number of smaller CAS by forming their product, it closes the cycle only with the very last application of the product operator: until that moment, the "cycle" still looks to the compiler as an open-ended chain of buffered channels. Because new data items can freely flow into it, such an open-ended chain can have a data item in any buffer at any time. Consequently, the CA for the largest chain (i.e., the chain of $k-1$ buffered channels, just before it becomes closed) has 2^{k-1} states. Only when $\texttt{Compiler}_{\text{centr}}$ forms the product of [the CA of the k-th buffered channel] and [the previously formed CA for the chain of $k-1$ buffered channels], the state space of 2^{k-1} states collapses into k states, as the compiler "finds out" that the open-ended chain is actually an input-closed cycle with exactly one data item. Clearly, because Sequencer$_k$ constitutes EarlyAsyncOutSequencer$_k$, also EarlyAsyncOutSequencer$_k$ itself suffers from this problem. A similar argument applies to Lock$_k$.

Thus, even for k-parametric connector families whose big CAS grow only linearly in k, $\texttt{Compiler}_{\text{centr}}$ can have scalability issues because of exponential growth in intermediate products. $\texttt{Compiler}_{\text{hybr}}$ has no problems with the kind of cycle-based exponential growth discussed above because of how it deals with such cycles in its partitioning algorithm.

5.3. Conclusion

For the four k-parametric connector families whose big CAS grow exponentially in k (i.e., EarlyAsyncBarrierMerger, EarlyAsyncMerger, LateAsyncReplicator, and LateAsyncRouter), hybrid compilation has clear advantages over centralized compilation, as we already expected. For the two k-parametric connector families whose big CAS and intermediate products grow only linearly in k (i.e., LateAsyncMerger and EarlyAsyncReplicator), centralized-approach compilation and hybrid-approach compilation do not make much of a difference; here, run-time performance—investigated in Section 6—becomes the key factor in deciding which approach to apply. For Alternator, centralized compilation has clear advantages over hybrid compilation. Finally, for the two k-parametric connector families whose intermediate products grow exponentially in k (i.e., EarlyAsyncOutSequencer and Lock), *at first sight*, hybrid compilation seems to have clear advantages over centralized compilation as suggested by Fig. 8. However, our previous analysis *also* showed that the big CAS—the only CAS that we actually care about—for both EarlyAsyncOutSequencer and Lock grow only linearly in k: if we can develop technology that enables $\texttt{Compiler}_{\text{centr}}$ to avoid temporary exponential growth of intermediate products, thus, $\texttt{Compiler}_{\text{centr}}$ should perform similar to $\texttt{Compiler}_{\text{hybr}}$.

One option is to equip $\texttt{Compiler}_{\text{centr}}$ with a novel static analysis technique to infer, *before* computing the full product, which states are unreachable *after* computing the full product. For instance, in the case of EarlyAsyncOutSequencer$_k$ (or its subconnector Sequencer$_k$), every state where two or more buffers contain a data item is unreachable in the full product but not so yet in the intermediate products. If $\texttt{Compiler}_{\text{centr}}$ can determine such "eventually unreachable states" from the start, it can already remove those states *while* computing the full product to keep the intermediate products as small as possible. Recently, we formulated this optimization technique and proved its correctness [33].

Another option is not really a solution to our problem but a way to avoid it. We observed that the Sequencer$_k$ subconnector of EarlyAsyncOutSequencer$_k$ causes its intermediate products to grow exponentially in k. For simplicity, let us therefore focus on this problematic Sequencer$_k$. The obvious way to construct a connector with the behavior of Sequencer$_k$ is by composing k buffered channels in a cycle, as we did before. An alternative way to construct such a connector, however, is by connecting a Sequencer$_{0.5k}$ to another Sequencer$_{0.5k}$ with a "glue subconnector". The details of this glue subconnector do not matter here: what matters is that in this alternative construction, $\texttt{Compiler}_{\text{centr}}$ can first compute the products of the Sequencer$_{0.5k}$ subconnectors to get two CAS with $0.5k$ states, and then compute the products of those CAS and the two-state CA of the glue subconnector. The largest intermediate CA encountered by the compiler during this process has at most $\max(2^{0.5k}, 0.5k \cdot 0.5k \cdot 2)$ states. In contrast, the largest intermediate CA for the obviously constructed Sequencer$_k$—the one with the cycle—has 2^{k-1} states. This analysis shows that *hierarchically* constructing Sequencer$_k$ out of Sequencer$_{l<k}$ subconnectors reduces its centralized-approach compilation complexity compared to its *flat* design. Admittedly, hierarchical design of a Sequencer$_k$ obfuscates its simple cyclic flat structure parameterized by k. Nevertheless, the above analysis justifies encouraging programmers to design connectors as hierarchically as possible.
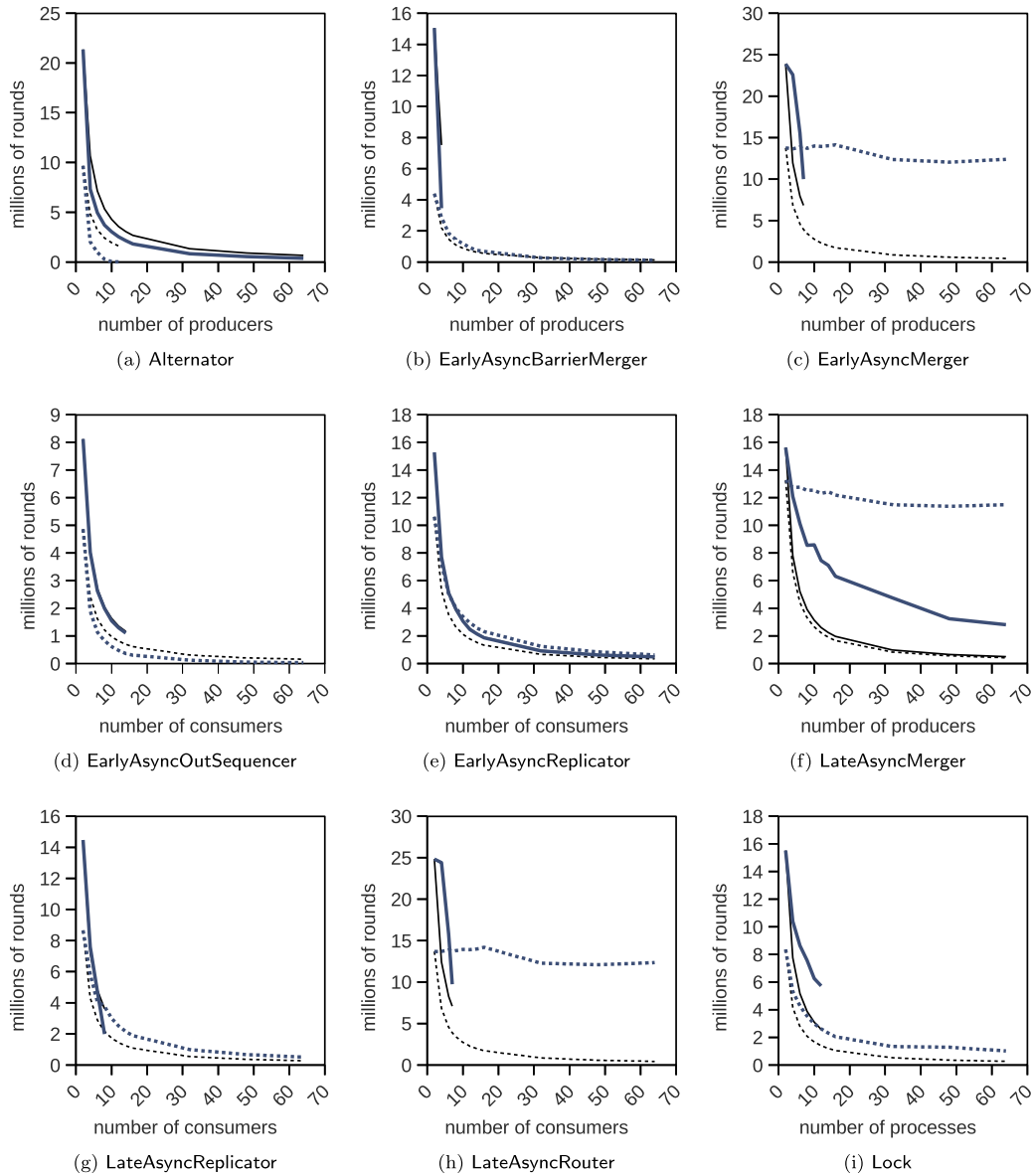
Fig. 10. Performance, in rounds per four minutes (solid/dotted lines for centralized/hybrid; thinner black curves for inverse-proportional growth).

6. Experimental results: execution

6.1. Measurements

We ran every successfully compiled connector with "empty" processes: in every iteration of its infinite loop, a producer/consumer had no work and immediately performed an I/O operation on its accessible boundary node. As a result, we measured the performance of only the compiler-generated code. Fig. 10 shows our measurements, in completed protocol rounds per four minutes. By dividing this number of rounds by 240, one gets the round-throughput, in rounds per second. By further dividing this number by the number of transitions per round, one gets the (transition-)throughput. Thinner black curves indicate, for every connector family, inverse-proportional growth (ipg) relative to the $k = 1$ measurement ($ipg(k) = meas(1)/k$), as a baseline to scalability; ideally, all measurements are *above* the inverse-proportional growth curve (e.g., if k doubles, connector overhead should ideally less than double, i.e., the number of completed rounds for $2k$ should be more than $\frac{1}{2}$ the number of completed rounds for k).

Figs. 10a, 10b, 10c, and 10f show the performances in the k-producers-single-consumer category. For LateAsyncMerger, EarlyAsyncMerger, and EarlyAsyncBarrierMerger, their centralized implementations outperform their hybrid implementations in cases involving only few producers (up to/including four in the case of LateAsyncMerger and EarlyAsyncBarrierMerger;

up to/including six in the case of EarlyAsyncMerger). In cases involving more producers, either the hybrid implementations outperform the centralized implementations, or $\texttt{Compiler}_\texttt{centr}$ failed to compile such that we cannot make a direct 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 worse than their corresponding hybrid counterparts. For Alternator, in contrast, its centralized implementations always outperform its hybrid implementations.

Figs. 10d, 10e, 10g, and 10h show the performances in the single-producer-k-consumers category. The figures for LateAsyncReplicator and LateAsyncRouter are similar to those of LateAsyncMerger, EarlyAsyncMerger, and EarlyAsyncBarrierMerger that we saw before: with only few consumers, their centralized implementations outperform their hybrid implementations, while with more consumers, their hybrid implementations outperform their centralized implementations. For EarlyAsyncReplicator, the performances of its centralized and hybrid implementations are nearly the same. For EarlyAsyncOutSequencer, because $\texttt{Compiler}_\texttt{centr}$ failed to generate code for $k > 14$, the comparison remains inconclusive.

6.2. Discussion

For six of the nine connector families, the obtained results look as expected. For those families, we observe that with low values of k (i.e., little parallelism), their centralized implementations outperform their hybrid implementations. In those cases, the increased throughput of hybrid implementations as compared to their centralized counterparts cannot yet compensate for their increased latency. As k increases and more parallelism becomes available, however, hybrid implementations start to outperform centralized implementations. In those cases, increased throughput does seem to compensate for increased latency. This, however, is not the only reason why hybrid implementations outperform centralized implementations for larger values of k. More importantly, we found that the latency of not only hybrid implementations but also centralized implementations increases with k. In fact, the latency of centralized implementations increases much more dramatically. By analyzing the big CAS computed by $\texttt{Compiler}_\texttt{centr}$ for the families currently under discussion, we found that their exponential growth (cf. Section 5) causes this steep increase in latency: the more transitions a CA has per state, the more time it takes to select and check any one of them at run-time. (EarlyAsyncReplicator constitutes a special case, where increased throughput and increased latency roughly balance out.)

Contrasting the families discussed in the previous paragraph, the results obtained for Alternator, EarlyAsyncOutSequencer, and Lock are more peculiar. In Section 5, we already briefly explained why $\texttt{Compiler}_\texttt{centr}$ succeeded in generating code for Alternator$_{k>12}$, while $\texttt{Compiler}_\texttt{hybr}$ failed. This, however, does not yet explain why centralized implementations of Alternator connectors outperform their hybrid implementations also at run-time. The reason becomes clear when we realize that Alternator$_k$ essentially behaves sequentially: in every round, the producers start by synchronously $\texttt{write}$ing their data items (and the consumer synchronously $\texttt{takes}$ the first data item), after which the consumer asynchronously $\texttt{takes}$ the remaining $k-1$ data items in sequence. The centralized implementation of Alternator$_k$ at run-time sequentially simulates one CA, which consists of k transitions between k states, that represents exactly this sequentiality, as shown in Fig. 9b. Its hybrid implementation, in contrast, at run-time uses k Java threads and, as such, suffers from *overparallelization*: it uses parallelism—and incurs the overhead that parallelism involves—to implement intrinsically sequential behavior. Because also EarlyAsyncOutSequencer and Lock essentially behave sequentially, they suffer from the same problem. For these two families, however, this observation is even more important than for Alternator. After all, hybrid-approach compilation fails for Alternator$_{k>12}$, so for larger k, we must use centralized-approach compilation anyway. For EarlyAsyncOutSequencer$_{k>14}$ and Lock$_{k>12}$, in contrast, centralized-approach compilation fails, even though centralized implementations of those connectors are, by extrapolation, likely to perform better than their hybrid counterparts.

Our centralized implementations consist of only one Java thread, which can do only one thing at a time. If many processes each perform an I/O operation roughly simultaneously, depending on the connector, this may result in contention (i.e., every process must wait until the protocol has time to handle the I/O operation of that process). To further study the effect of contention, we repeated our experiments with z-parametric producers/consumers that wait a random amount of time between 0 and [z times the previously measured round-latency] before they perform their $\texttt{write/take}$, for $z \in \{1, 10, 100\}$. Our measurements appear elsewhere, in a technical report [34]. The short conclusion is that as z increases, the performance of centralized implementations and hybrid implementations becomes more similar. We doubt whether this can be ascribed to less contention, though. Instead, we consider it more likely that the producers'/consumers' waiting times now dominate our measurements. Although perhaps not too surprising, we nevertheless consider this something that one should be aware of: the more work processes perform, the less important the choice between centralized/hybrid implementation becomes (with respect to run-time performance).

6.3. Conclusion

For six of the nine connector families, the obtained results are as we expected: their centralized implementations outperform their hybrid implementations for smaller values of k, while their hybrid implementations outperform their centralized implementations for larger values of k. As k increases and more parallelism becomes available, the higher throughput of hybrid implementations as compared to their centralized counterparts compensates for their higher latency, while the latency of centralized implementations dramatically increases.

Because Alternator, EarlyAsyncOutSequencer, and Lock essentially behave sequentially, their centralized implementations in fact outperform their hybrid implementations for all k. This is a strong incentive to improve our centralized-approach compilation technology (e.g., the optimization techniques proposed in Section 5.3).

7. General framework to improve `Compiler`~hybr~

7.1. Motivation

Through our experimental results in Sections 5 and 6, we discovered the following two issues with the hybrid approach: at build-time, hybrid-approach compilation may suffer from transition relation explosion, while at run-time, hybrid-approach implementations may suffer from overparallelization. Alternator, for instance, has both these issues. Our current hybrid-approach compiler $\mathtt{Compiler}_{\mathrm{hybr}}$, thus, does not yet strike the perfect middle ground between the centralized approach and the distributed approach (as explained in Section 1.2). In particular, in terms of the connector compilation spectrum in Figs. 2 and 7, $\mathtt{Compiler}_{\mathrm{hybr}}$ is still too far to the right: to avoid transition relation explosion, as explained in Section 5, $\mathtt{Compiler}_{\mathrm{hybr}}$ should compute "larger-than-medium" CAs (to filter out permanently disabled transitions), while to avoid overparallelization, $\mathtt{Compiler}_{\mathrm{hybr}}$ should introduce more sequentiality in its generated code. In this section, we present a general framework to move $\mathtt{Compiler}_{\mathrm{hybr}}$ further toward the left of the compilation spectrum.

7.2. Idea

Using $\mathtt{Compiler}_{\mathrm{hybr}}$, as explained in Section 3, every medium CA has its own Java thread at run-time. Overparallelization means that (some of) the parallelism among those threads is actually *useless* (if not counterproductive). The central question to avoid overparallelization, then, is how to preserve only *useful* parallelism. This, in turn, raises the question of when parallelism is useful. To answer this question, first, observe that at least the parallelism among Java threads for "boundary CAs" (i.e., medium CAs with boundary nodes accessible to processes) is useful. Such parallelism, after all, allows I/O operations of any process to complete independently of, and in parallel with, I/O operations of other processes. As processes perform the real computations of an application—protocols are, essentially, just overhead—independence and parallelism among them is desirable. In contrast, parallelism among "internal CAs" (i.e., medium CAs without boundary nodes) is often actually useless, in the sense that their execution in separate Java threads increases latency (because of the necessary consensus algorithm) typically without increasing throughput (because parallelism among processes is affected only by boundary CAs). For instance, the parallelism between the medium CAs for the two asynchronous regions in Alternator$_3$ in Fig. 9 would be useful only if this parallelism would allow (the processes connected to) the synchronous regions connected to these asynchronous regions to progress independently of each other. However, these synchronous regions are, in fact, *the same* single synchronous region. This makes the "parallelism" between the medium CAs for the two asynchronous regions effectively useless: their neighboring medium CA for the single synchronous region can, after all, never fire a transition independently, in parallel, of itself.

To preserve only useful parallelism (among boundary CAs), the idea is to *merge* synchronous and asynchronous regions together, until only "boundary synchronous regions" remain, separated by asynchronous regions. Intuitively, merging regions in this way mitigates transition relation explosion at build-time because a compiler now essentially computes larger-than-medium CAs (which may eliminate permanently disabled transitions), while overparallelization at run-time is mitigated because the compiler preserves only useful parallelism among boundary regions. As with all our compilation technology, we formulate such merging of regions not specifically in terms of Reo but generally in terms of (medium) CAs, as follows.

7.3. Framework

Suppose that, after partitioning and computing per-subset products, we have m medium CAs $\boldsymbol{b}_1, \dots, \boldsymbol{b}_m$ (as in Fig. 7). Now, instead of directly generating code for those medium CAs (as our current hybrid-approach compiler does), we first compute a *graph representation* of the dependencies among those medium CAs. To do this, let $\{B, B^{\mathrm{async}}\}$ denote a partition of $\{\boldsymbol{b}_1, \dots, \boldsymbol{b}_m\}$, where B^{async} contains only asynchronous CAs (as defined directly below Definition 1), and where $B = \{\boldsymbol{b}_1, \dots, \boldsymbol{b}_m\} \setminus B^{\mathrm{async}}$ denotes the remaining CAs. A *dependency graph* for $\{B, B^{\mathrm{async}}\}$, then, is a directed graph with vertices $B \cup B^{\mathrm{async}}$ and with an arc from $\boldsymbol{b}^{\mathrm{async}} \in B^{\mathrm{async}}$ to $\boldsymbol{b} \in B$ whenever $\boldsymbol{b}^{\mathrm{async}}$ and $\boldsymbol{b}$ share at least one end (i.e., $\boldsymbol{b}^{\mathrm{async}}$ and $\boldsymbol{b}$ correspond to neighboring regions). These arcs may also be interpreted as undirected edges, yielding a bipartite graph with partite sets B and B^{async}. Let $\mathrm{End}(\boldsymbol{b})$ denote the set of ends of a CA $\boldsymbol{b}$.

Definition 3. A dependency graph is a tuple $(B, B^{\mathrm{async}}, \mapsto)$, where:

- $B \subseteq \mathbb{A}\textsc{utom}$ denotes a set of CAs
- $B^{\mathrm{async}} \subseteq \mathbb{A}\textsc{utom}$ denotes a set of a CAs such that $\left[\xrightarrow{1} (\boldsymbol{b}^{\mathrm{async}}) \ \textbf{for all} \ \boldsymbol{b}^{\mathrm{async}} \in B^{\mathrm{async}} \right]$
- $\mapsto \, \subseteq B^{\mathrm{async}} \times B$ denotes the smallest relation induced by the following rule:
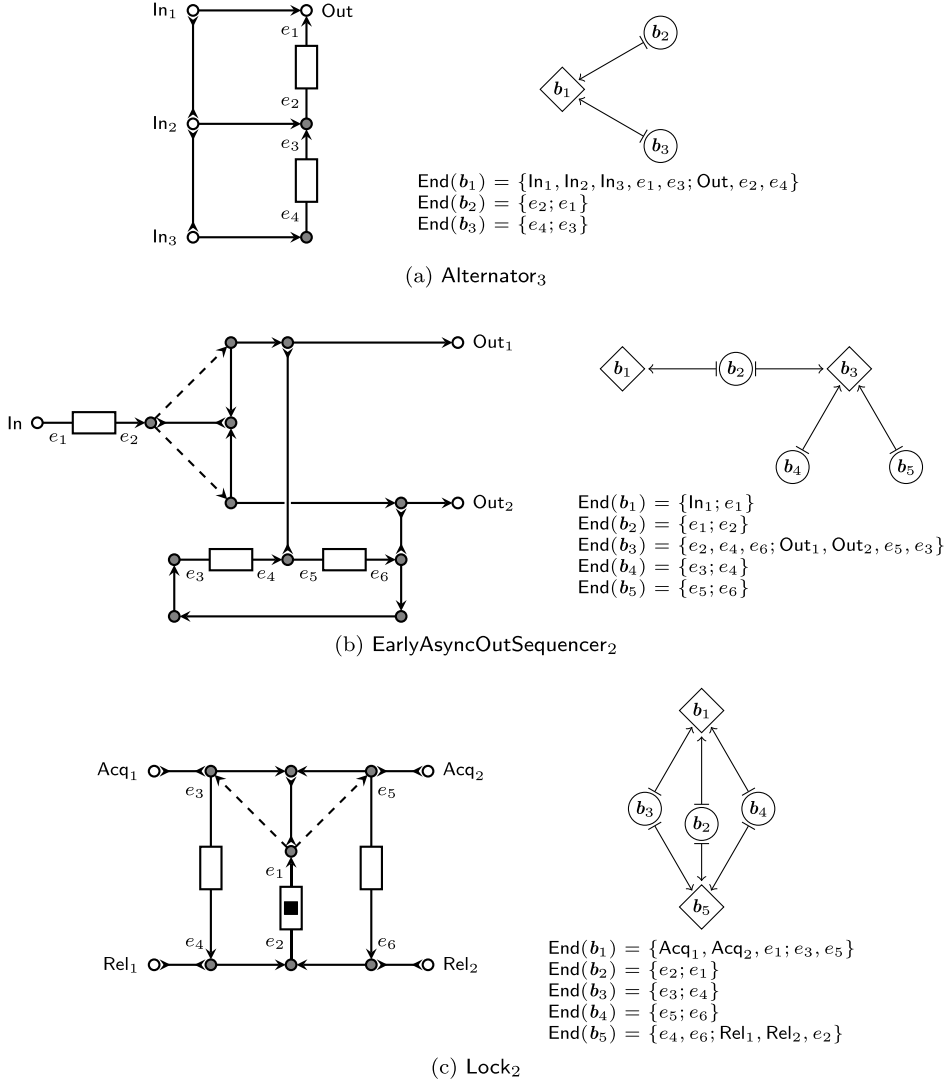
$$\mathrm{End}(b_1) = \{\mathrm{In}_1, \mathrm{In}_2, \mathrm{In}_3, e_1, e_3; \mathrm{Out}, e_2, e_4\}$$
$$\mathrm{End}(b_2) = \{e_2; e_1\}$$
$$\mathrm{End}(b_3) = \{e_4; e_3\}$$

(a) Alternator$_3$

$$\mathrm{End}(b_1) = \{\mathrm{In}_1; e_1\}$$
$$\mathrm{End}(b_2) = \{e_1; e_2\}$$
$$\mathrm{End}(b_3) = \{e_2, e_4, e_6; \mathrm{Out}_1, \mathrm{Out}_2, e_5, e_3\}$$
$$\mathrm{End}(b_4) = \{e_3; e_4\}$$
$$\mathrm{End}(b_5) = \{e_5; e_6\}$$

(b) EarlyAsyncOutSequencer$_2$

$$\mathrm{End}(b_1) = \{\mathrm{Acq}_1, \mathrm{Acq}_2, e_1; e_3, e_5\}$$
$$\mathrm{End}(b_2) = \{e_2; e_1\}$$
$$\mathrm{End}(b_3) = \{e_3; e_4\}$$
$$\mathrm{End}(b_4) = \{e_5; e_6\}$$
$$\mathrm{End}(b_5) = \{e_4, e_6; \mathrm{Rel}_1, \mathrm{Rel}_2, e_2\}$$

(c) Lock$_2$

Fig. 11. Example dependency graphs, where diamonds represent vertices in B, and where circles represent vertices in B^{async}.

$$\frac{b^{\mathrm{async}} \in B^{\mathrm{async}} \ \ \textbf{and} \ \ b \in B \ \ \textbf{and} \ \ \mathrm{End}(b^{\mathrm{async}}) \cap \mathrm{End}(b) \neq \emptyset}{b^{\mathrm{async}} \mapsto b}$$

$\mathbb{G}$RAPH denotes the set of all dependency graphs. □

Fig. 11 shows examples of dependency graphs.

Next, we define a *contraction* operator on dependency graphs to merge synchronous and asynchronous regions in terms of dependency graphs (to mitigate transition relation explosion and overparallelization). This contraction operator has three operands: a CA b, called the *gobbler*, a CA b^{async}, called the *gobblee*, and a dependency graph $G = (B, B^{\mathrm{async}}, \mapsto)$. There are two cases to consider:

- If B contains gobbler b (i.e., b corresponds to a synchronous region), *and* if B^{async} contains gobblee b^{async} (i.e., b^{async} corresponds to an asynchronous region), *and* if b^{async} depends on b according to $\mapsto$ (i.e., b^{async} and b correspond to neighboring regions), then the contraction operator produces a new dependency graph by letting gobbler b "gobble up" gobblee b^{async} in G (i.e., merge the regions corresponding to b and b^{async}).
  The naive way to define such "gobbling" is by removing b from B, by removing b^{async} from B^{async}, and by adding the product $b \otimes b^{\mathrm{async}}$ to B (i.e., the two existing regions constitute a new synchronous region when merged). The problem with this approach is that it is unclear how to redefine $\mapsto$ after gobbling: if b^{async} depends on other CAs beside b

before gobbling, then these dependencies need to be preserved after gobbling, but the only way to achieve this is by letting $\boldsymbol{b} \otimes \boldsymbol{b}^{\text{async}}$ depend on these CAs, which Definition 3 forbids (because $\boldsymbol{b} \otimes \boldsymbol{b}^{\text{async}} \in B$).

A better way to define "gobbling" is therefore to let $\boldsymbol{b}$ not only gobble up $\boldsymbol{b}^{\text{async}}$, henceforth called the *primary gobblee*, but also all CAs in B on which $\boldsymbol{b}^{\text{async}}$ depends according to $\mapsto$ (beside $\boldsymbol{b}$), called *secondary gobblees*. This approach ensures that there are no "dangling" dependencies after gobbling.

For notational convenience in formulating a contraction algorithm (presented shortly), the contraction operator not only outputs the updated dependency graph but also the gobbler after gobbling up its gobblees (both primary and secondary) and the set of secondary gobblees.

- Otherwise, if B does not contain gobbler $\boldsymbol{b}$, *or* if B^{async} does not contain gobblee $\boldsymbol{b}^{\text{async}}$, *or* if $\boldsymbol{b}^{\text{async}}$ does not depend on $\boldsymbol{b}$ according to $\mapsto$, then the contraction operator does nothing.

In the following definition, $\boldsymbol{b}'$ denotes the gobbler after gobbling up its gobblees (both primary and secondary), while B' denotes the set of secondary gobblees.

Definition 4. $\mathbb{G} : \text{AUTOM} \times \text{AUTOM} \times \text{GRAPH} \to \text{AUTOM} \times 2^{\text{AUTOM}} \times \text{GRAPH}$ denotes the function defined by the following equation:

$$
\boldsymbol{b} \,\mathbb{G}\, \boldsymbol{b}^{\text{async}} : (B, B^{\text{async}}, \mapsto) = \begin{cases} \left(\boldsymbol{b}', B', \begin{pmatrix} (B \setminus B') \cup \{\boldsymbol{b}'\}, \\ B^{\text{async}} \setminus \{\boldsymbol{b}^{\text{async}}\}, \\ \mapsto' \end{pmatrix}\right) & \text{if } \begin{bmatrix} \boldsymbol{b} \in B \\ \text{and } \boldsymbol{b}^{\text{async}} \in B^{\text{async}} \\ \text{and } \boldsymbol{b}^{\text{async}} \mapsto \boldsymbol{b} \end{bmatrix} \\ \\ (\boldsymbol{b}, \emptyset, (B, B^{\text{async}}, \mapsto)) & \textbf{otherwise} \end{cases}
$$

where $\boldsymbol{b}' = \boldsymbol{b}^{\text{async}} \otimes \bigotimes B'$, $B' = \{\hat{\boldsymbol{b}} \mid \boldsymbol{b}^{\text{async}} \mapsto \hat{\boldsymbol{b}}\}$, and $\mapsto'$ is the smallest relation induced by the following rules:

$$
\frac{\hat{\boldsymbol{b}}^{\text{async}} \mapsto \hat{\boldsymbol{b}} \text{ and } \hat{\boldsymbol{b}} \notin B'}{\hat{\boldsymbol{b}}^{\text{async}} \mapsto' \hat{\boldsymbol{b}}} \qquad \frac{\hat{\boldsymbol{b}}^{\text{async}} \mapsto \hat{\boldsymbol{b}} \text{ and } \hat{\boldsymbol{b}} \in B' \text{ and } \hat{\boldsymbol{b}}^{\text{async}} \neq \boldsymbol{b}^{\text{async}}}{\hat{\boldsymbol{b}}^{\text{async}} \mapsto' \boldsymbol{b}'} \qquad \square
$$

The notation $\boldsymbol{b} \,\mathbb{G}\, \boldsymbol{b}^{\text{async}} : G$ should be read, from left to right, as "gobbler $\boldsymbol{b}$ gobbles up ($\mathbb{G}$) the gobblee $\boldsymbol{b}^{\text{async}}$ in (:) dependency graph G".

Given this contraction operator, the idea is to let boundary CAs gobble up internal CAs until only boundary CAs remain, separated by internal asynchronous CAs. Specifically, we let a boundary CA gobble up a number of internal CAs only if none of the secondary gobblees is a boundary CA different from the gobbler. Otherwise, in terms of Reo, we would merge different boundary regions into one, thereby losing useful parallelism.

To formulate this approach as an algorithm, let $B^{\text{bnd}} \subseteq B$ denote the set of boundary CAs in a dependency graph $(B, B^{\text{async}}, \mapsto)$. Associate a unique natural number—an *id*—to every boundary CA in B^{bnd}, and denote the corresponding (partial) bijection by $\iota : \mathbb{N} \rightharpoonup B^{\text{bnd}}$. Let ι^{-1} denote the inverse of ι. To indicate that a boundary CA $\boldsymbol{b} \in B^{\text{bnd}}$, with id $n = \iota^{-1}(\boldsymbol{b})$, needs to gobble up an asynchronous CA $\boldsymbol{b}^{\text{async}}$, we write the pair $(n, \boldsymbol{b}^{\text{async}})$. A sequence of such pairs, then, naturally corresponds to a sequence in which boundary CAs need to gobble up internal CAs. Formally, we represent such a sequence with a total order on $\text{Dom}(\iota) \times B^{\text{async}}$, denoted by $<$. For instance, $(1, \boldsymbol{b}_2) < (3, \boldsymbol{b}_4) < (3, \boldsymbol{b}_5)$ means that first the boundary CA with id 1 must gobble up $\boldsymbol{b}_2$. Subsequently, the boundary CA with id 3 must gobble up $\boldsymbol{b}_4$. Finally, the boundary CA with id 3 must gobble up $\boldsymbol{b}_5$. Note that bijectivity is a property of ι and not of $<$. Thus, it is no problem that id 3 occurs twice in $<$; it simply means that the boundary CA with id 3 must gobble twice. In contrast, two boundary CAs *cannot* have the same id, because this violates the bijectivity of ι.

Fig. 12 shows a *contraction algorithm*, which consumes $(B, B^{\text{async}}, \mapsto)$ and ι and $<$ as input and produces a contracted dependency graph as output. This algorithm terminates because k is finite and j is increasing. Because $\mathbb{G}$ produces only well-formed dependency graphs by Definition 4 (i.e., with abuse of terminology, we can say that "GRAPH is closed under $\mathbb{G}$"), the final G after the algorithm terminates is also a well-formed dependency graph (i.e., $G \in \text{GRAPH}$).

The output of our algorithm depends on order $<$. Consider, for instance, the connector in Fig. 13. In this example, we can let either the left-most boundary CA ($\boldsymbol{b}_1$) or the right-most boundary CA ($\boldsymbol{b}_5$) gobble up the internal CAs, but not both, with different results: in the former case, we end up with $\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$, $\boldsymbol{b}_4$, and $\boldsymbol{b}_5$, whereas in the latter case, we end up with $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, and $\boldsymbol{b}_3 \otimes \boldsymbol{b}_4 \otimes \boldsymbol{b}_5$. Thus, although the algorithm in Fig. 12 itself is deterministic, it can produce different results depending on input $<$. Every result (i.e., every $<$) may lead to a different set of performance characteristics at run-time. A thorough investigation into finding an optimal $<$ (to maximize performance) given the shape of a dependency graph (and/or properties of the CAs involved) is beyond the scope of this paper. On the contrary, the main contribution of this section is a general framework for merging synchronous and asynchronous regions (i.e., Definitions 3 and 4 and the algorithm in Fig. 12) that indeed allows adoption of alternative orders and their respective performance characteristics.

In the rest of this section, in examples to illustrate our algorithm, we compute $<$ according to the simple heuristic that all boundary CAs gobble up internal CAs in a round-robin fashion (see the examples below), one after the other, starting from boundary CAs with source nodes. The idea behind this heuristic is to distribute the work of internal CAs evenly over

Input:

- $(B, B^{\text{async}}, \mapsto) \in \mathbb{G}\text{RAPH}$ such that $B^{\text{bnd}} \subseteq B$
- $\iota : \mathbb{N} \rightharpoonup B^{\text{bnd}}$ such that ι is bijective
- $< \;\subseteq \text{Dom}(\iota) \times B^{\text{async}}$ such that:

$$\Big[< \text{ is a strict total order}\Big] \text{ and } \begin{bmatrix} (n_1, \boldsymbol{b}_1) < \cdots < (n_k, \boldsymbol{b}_k) \;\; \textbf{for some } \; n_1, \ldots, n_k \in \text{Dom}(\iota) \\ \text{and } \; \boldsymbol{b}_1, \ldots, \boldsymbol{b}_k \in B^{\text{async}} \end{bmatrix}$$

Algorithm:

1. ∥ Initialize return variable G
   $G := (B, B^{\text{async}}, \mapsto)$
2. ∥ Initialize loop variable j
   $j := 1$
3. ∥ For each pair $(n_j, \boldsymbol{b}_j)$ in chain $(n_1, \boldsymbol{b}_1) < \cdots < (n_k, \boldsymbol{b}_k)$
   `while` $j \leq k$ `do`
   (a) ∥ Initialize variable $\boldsymbol{b}$ as the current boundary automaton associated with id n_j
       $\boldsymbol{b} := \iota(n_j)$
   (b) ∥ Tentatively contract
       $(\boldsymbol{b}', B', G') := \boldsymbol{b} \circlearrowleft \boldsymbol{b}_j : G$
   (c) ∥ If gobbler $\boldsymbol{b}$ did not gobble up another boundary CA, finalize contraction
       `if` $B' \cap (B^{\text{bnd}} \setminus \{\boldsymbol{b}\}) = \emptyset$ `then` $G := G'$
   (d) ∥ Update the boundary automaton associated with id n_j
       $\iota := \iota[n_j \mapsto \boldsymbol{b}']$
   (e) ∥ Update loop variable j
       $j := j + 1$

Output:

- $G \in \mathbb{G}\text{RAPH}$

Fig. 12. Contraction algorithm.
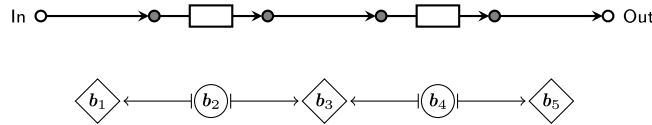
Fig. 13. Example connector (and its dependency graph) to show that the result of the contraction algorithm in Fig. 12 depends on input $<$.

all boundary CAs. It is possible to formally define this heuristic, but because this does not yield any fundamental new insight, we skip this here; in practice, it is straightforward to implement this heuristic with nested loops (the top one of which iterates over boundary CAs; the nested one of which iterates over internal CAs), possibly on-the-fly during the execution of the contraction algorithm (instead of precomputing $<$). In future work, as mentioned above, we need to study whether it is possible to compute an optimal $<$ (on-line or off-line), under some formal definition of optimality; such a definition currently does not exist. It is also interesting to investigate to what extent our current straightforward heuristic approximates such a theoretical optimum.

7.4. Examples

- Alternator$_3$
  Reconsider Fig. 11a. Observe that $\boldsymbol{b}_1$ is a boundary CA. Define $\iota = \{(1, \boldsymbol{b}_1)\}$ (i.e., $\boldsymbol{a}_1$ has id 1).
  Order $<$ can now be computed (automatically) using the heuristic explained above, with nested loops. In the outer loop, we iterate over all boundary CAs. Because there is only one boundary CA (namely $\boldsymbol{a}_1$), this loop has only one iteration. In the inner loop, we iterate over all internal CAs. For every internal CA $\boldsymbol{a}_j$, we add a new pair $(1, \boldsymbol{a}_j)$ to "the back" of the order. In this way, we get $(1, \boldsymbol{b}_2) < (1, \boldsymbol{b}_3)$. This means that $\boldsymbol{a}_1$ (i.e., the CA with id 1) first gobbles up $\boldsymbol{a}_2$ and then $\boldsymbol{a}_3$.
  Fig. 14a shows the evolution of the dependency graph in Fig. 11a under the algorithm in Fig. 12. Every $\Rightarrow$ represents the execution of one iteration of the `while` loop.
  Applied to the dependency graph for Alternator$_3$, our algorithm yields a dependency graph consisting of one comprehensive boundary CA, namely, the very same big CA as computed in the centralized approach. This observation generalizes to Alternator$_k$. For Alternator$_k$, thus, our algorithm reduces the hybrid approach to the centralized approach, which is

(a) $\mathsf{Alternator}_3$ (see also Figure 11a)

(b) $\mathsf{EarlyAsyncOutSequencer}_2$ (see also Figure 11b)
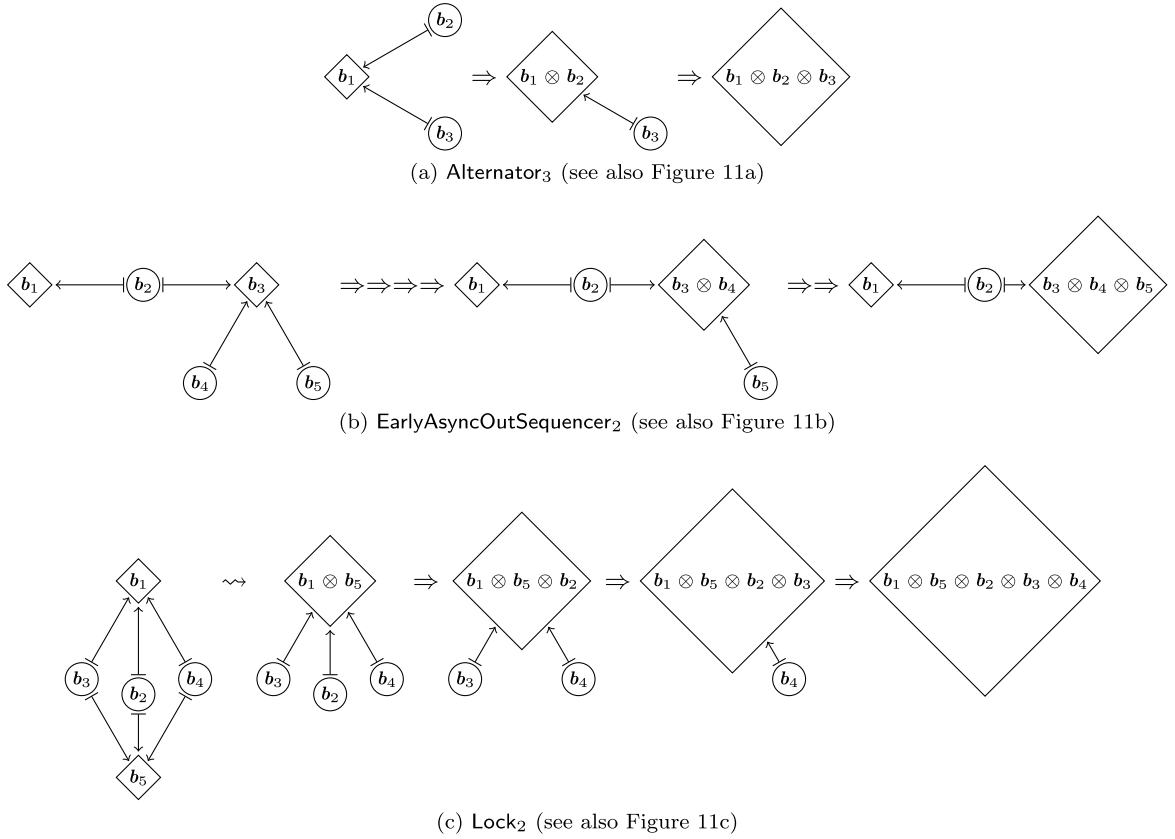

(c) Lock_2 (see also Figure 11c)

Fig. 14. Examples of the contraction algorithm in Fig. 12.

exactly what we want in order to avoid transition relation explosion at build-time (as explained in Section 5.2) and overparallelization at run-time (as explained in Section 6.2).

- $\mathsf{EarlyAsyncOutSequencer}_2$

Reconsider Fig. 11b. Observe that $\boldsymbol{b}_1$ and $\boldsymbol{b}_3$ are boundary CAs. Define $\iota = \{(1, \boldsymbol{b}_1), (3, \boldsymbol{b}_3)\}$ (i.e., $\boldsymbol{a}_1$ has id 1, and $\boldsymbol{a}_3$ has id 3).

Order $<$ can now be computed (automatically) using the heuristic explained above, with nested loops. In the outer loop, we iterate over all boundary CAs. In each iteration, we create an auxiliary order; after the outer loop, we merge these auxiliary orders into the final $<$. In the first iteration of the outer loop, we consider $\boldsymbol{a}_1$. In the inner loop, we iterate over all internal CAs. For every internal CA $\boldsymbol{a}_j$, we add a new pair $(1, \boldsymbol{a}_j)$ to "the back" of the auxiliary order. In this way, we get $(1, \boldsymbol{b}_2) <_1 (1, \boldsymbol{b}_4) <_1 (1, \boldsymbol{b}_5)$, where $<_1$ denotes the auxiliary order. In the second iteration of the outer loop, we consider $\boldsymbol{a}_3$. In the same way as in the first iteration, we get $(3, \boldsymbol{b}_2) <_3 (3, \boldsymbol{b}_4) <_3 (3, \boldsymbol{b}_5)$, where $<_3$ denotes the auxiliary order. By "zipping" $<_1$ and $<_3$, we get $(1, \boldsymbol{b}_2) < (3, \boldsymbol{b}_2) < (1, \boldsymbol{b}_4) < (3, \boldsymbol{b}_4) < (1, \boldsymbol{b}_5) < (3, \boldsymbol{b}_5)$.

Fig. 14b shows the evolution of the dependency graph in Fig. 11b under the algorithm in Fig. 12. Consecutive $\Rightarrow$-symbols represent that the dependency graph does not change during all but the last iteration represented by the $\Rightarrow$-symbols. For instance, the dependency graph does not change in the first iteration, because contraction $\iota(1) \circlearrowright \boldsymbol{b}_2 : G$ causes $\boldsymbol{b}_1$ to gobble up not only $\boldsymbol{b}_2$ but also $\boldsymbol{b}_3$, which is a boundary CA (and recall that boundary CAs have useful parallelism and should, therefore, not be composed at build-time); in Fig. 12, the corresponding check occurs in step 3c. As another example, the dependency graph does not change in the third iteration, because contraction $\iota(1) \circlearrowright \boldsymbol{b}_4$ causes $\boldsymbol{b}_1$ to gobble up $\boldsymbol{b}_4$, which is not a neighboring CA, so the contraction operator does nothing (see Definition 4).

Applied to the dependency graph for $\mathsf{EarlyAsyncOutSequencer}_2$, our algorithm yields a dependency graph with two boundary CAs—one of which is larger-than-medium—and one internal asynchronous CA between them. This observation generalizes to $\mathsf{EarlyAsyncOutSequencer}_k$. For $\mathsf{EarlyAsyncOutSequencer}_k$, thus, our algorithm moves the hybrid approach toward the left of the compilation spectrum in Figs. 2 and 7, thereby alleviating overparallelization at run-time (while maintaining useful parallelism between the two boundary CAs).

One concern is that by moving the hybrid approach toward the left of the compilation spectrum, $\mathsf{EarlyAsyncOutSequencer}_k$ becomes subject to the centralized-approach issue of "exponentially sized intermediate prod-

ucts". To resolve this, we can use the same optimization techniques as proposed in Section 5.3, one of which we already developed and implemented in a proof-of-concept tool [33].

- Lock$_2$

  Reconsider Lock$_2$ in Fig. 11c. Observe that $\boldsymbol{b}_1$ and $\boldsymbol{b}_5$ are boundary CAS. Define $\iota = \{(1, \boldsymbol{b}_1), (5, \boldsymbol{b}_5)\}$ (i.e., $\boldsymbol{b}_1$ has id 1, and $\boldsymbol{b}_5$ has id 5).

  Regardless of the definition of $<$, running the algorithm in Fig. 12 does not change the dependency graph in Fig. 11c whatsoever, because every possible contraction causes $\boldsymbol{b}_1$ to gobble up also $\boldsymbol{b}_5$ or vice versa. Crucially, however, observe that $\boldsymbol{b}_1$ and $\boldsymbol{b}_5$ are connected to the same processes (i.e., every i-th process has access both to node Acq$_i$ of $\boldsymbol{b}_1$ and to node Rel$_i$ of $\boldsymbol{b}_5$). This means that the parallelism among these two boundary CAS is actually useless and that $\boldsymbol{b}_1$ and $\boldsymbol{b}_5$ should be composed already at build-time.

  To resolve this issue, we need a *preprocessing step* before we run our algorithm, to compose boundary CAS that are connected to the same processes. To formalize this composition, we can introduce some formal representation of processes, for instance as CAS. Such a formalism is straightforward, but as it does not yield any new fundamental insight, we skip it here. Implementing such a preprocessing step in a compiler is also straightforward.

  Fig. 14c shows the evolution of the dependency graph in Fig. 11c (for an order $<$ computed in the same way as for Alternator$_3$, above). In this figure, $\rightsquigarrow$ represents the preprocessing step. The rest of the figure (and its analysis) is similar to Fig. 14a of Alternator$_3$. For Lock$_k$, thus, our algorithm reduces the hybrid approach to the centralized approach, thereby avoiding overparallelization at run-time; the remark about exponentially sized intermediate products in our previous discussion of EarlyAsyncOutSequencer$_k$ applies here, too, though.

7.5. Comparison with [33]

Among other findings, our experimental results in Sections 5 and 6 suggest that:

(1) Hybrid-approach compilation may suffer from exponentially sized CAS, caused by transition relation explosion.
(2) Hybrid implementations may overparallelize inherently sequential connectors.
(3) Centralized-approach compilation may suffer from exponentially sized intermediate products.

The general framework presented in this section aims to address findings (1) and (2). Concurrently, we also worked on a different optimization technique that addresses finding (3) [33].

The problem underlying finding (3) is that the centralized-approach compiler computes every full end product "one CA at a time" (i.e., first $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 = \boldsymbol{a}_{12}$, then $\boldsymbol{a}_{12} \otimes \boldsymbol{a}_3 = \boldsymbol{a}_{123}$, then $\boldsymbol{a}_{123} \otimes \boldsymbol{a}_4$, and so on). As a result, the state space of the intermediate products ($\boldsymbol{a}_{12}$, $\boldsymbol{a}_{123}$, and so on) may grow exponentially, even if many of those states become unreachable in the full product. The idea behind the [33]-optimization is to not generate such "eventually unreachable states" in the first place. To achieve this, instead of computing the full product "one CA at a time", the [33]-optimization ensures that the compiler computes the full product by processing all CAS at the same time: it starts from the initial state of the full product (which is the tuple consisting of the initial states of all constituent CAS), and computes successor states of this initial state by simultaneously considering the transition relations of all constituent CAS. This goes on until no new successor states are found. In this way, effectively, the compiler computes only the reachable states of the full product, without ever computing an "eventually unreachable state".

The [33]-optimization affects only compilation; it has no effect on run-time performance, because the full product is identical to the one computed without the [33]-optimization. Furthermore, the [33]-optimization cannot alleviate transition relation explosion (finding (1)), because transition relation explosion occurs in the full product. If the full product happens to contain (exponentially) many transitions, the [33]-optimization will not eliminate those. As such, the general framework presented in this section indeed serves a different purpose than the [33]-optimization. This general framework and the [33]-optimization can and should be used together, though (as already mentioned in the EarlyAsyncOutSequencer$_2$ example, above). The general framework presented in this section and the [33]-optimization are, thus, complementary.

8. Related work

8.1. Reo

Although we use Reo's CA semantics in our research on compilation [20,21], many other formal semantics exist [35], including those based on coalgebra [36], the connector coloring framework [5], Plotkin-style structural operational semantics [37], the tile model [38], and the Unifying Theories of Programming [39].

There is a striking visual resemblance between Reo connectors and Petri nets (e.g., [40]): places in Petri nets seem to correspond to asynchronous channels in Reo connectors, while transitions in Petri nets seem to correspond to synchronous channels and nodes in Reo connectors. The actual (non)correspondence between Reo connectors and Petri nets is, however, substantially more subtle than their visual similarities may suggest. While it is possible to translate a Petri net into a Reo connector (albeit using a somewhat more complicated encoding than the previously suggested correspondence between Petri net/Reo connector elements), there is no known structural translation of Reo connectors into Petri nets (i.e.,

Reo connectors seem more expressive). It *is* possible, though, to structurally translate Reo connectors into *zero-safe nets* [41]. Zero-safe nets constitute a variant of Petri nets with special execution semantics [42]. An interesting research area where Petri nets and Reo connectors meet is *scheduling*: recently, Dokter et al. have studied a game-theoretic framework for deriving schedules for processes coordinated by Reo connectors [43], not unlike the work on deriving schedules for applications modeled as Petri nets [44–47].

Beside Petri nets, Reo connectors may seem similar also to specifications in other higher-level graphical languages, such as Bpmn specifications and Uml activity diagrams. Arbab et al., Changizi et al., and Sun Meng et al. have worked on translating such specifications into Reo [24–26]. Also, Proença and Clarke studied the relation between Reo and orchestration language Orc [48], while Talcott et al. compared Reo with coordination languages Arc, and Pbrd [49].

8.2. Distributed coordination

In the mid 1980s, Gelernter introduced the coordination language Linda [50]. At the heart of Linda lies the concept of a *tuple space*, a structure in which both processes and tuples of data, originating from and accessible to those processes, "float". Although a tuple space gives the programmer the illusion of shared memory, at the hardware level, this memory may actually reside at n different locations. Several approaches to implementing physically distributed tuple spaces exist. For instance, one can maintain the entire tuple space at one of the n locations (e.g., Feng et al. [51], Wyckoff et al. [52]), but although simple to implement, this does not scale well in the number of processes [53]. The centralized approach in this paper actually has a similar scalability problem, as all parallelism is sequentialized. Alternatively, one can scatter (with or without replication) the tuples in the tuple space over all n locations. Although such an approach has better scalability, one must resolve several issues to obtain a workable implementation, such as deciding where to store which tuple, efficiently retrieving tuples, and load balancing [2]. For instance, Russello et al. developed an adaptive distributed tuple space middleware that monitors processes and, based on their monitored behavior, selects a distribution/replication policy for tuples that meets predefined performance/availability requirements best [54]. Other works on implementing distributed tuple spaces include the work by Bjornson [55], Feng et al. [53], Rowstron and Wood [56], Menezes and Tolksdorf [57], and Atkinson [58]. Although both distributed tuple spaces and the hybrid approach in this paper facilitate a form of distributed coordination, they differ in one fundamental aspect: whereas distributed tuple spaces distribute data (i.e., tuples), the hybrid approach distributes control (i.e., medium cas).

Bonakdarpour et al. worked on an approach for automatically generating distributed implementations for specifications in Bip [59], a framework for specifying component-based systems at three specification levels [29]: behavior of components, interaction between components (similar to transitions in cas), and priorities on interaction. Bip forbids simultaneous execution of conflicting interactions (similar to transitions in cas with overlapping synchronization constraints). In automatically-generated distributed implementations of Bip specifications, therefore, Bonakdarpour et al. have to ensure that such conflicting interactions execute mutually exclusively. To achieve 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 the top layer resolves potential conflicts. In terms of the hybrid approach in this paper, the bottom layer represents processes, while the middle layer roughly represents a product expression of cas. Importantly, however, Bonakdarpour et al. aim for a finer distribution granularity than we do, which requires them to handle conflicting interactions with their third layer at run-time. We avoid this problem in the hybrid approach, by putting cas with "conflicting transitions" in the same subset of the computed partition at build-time, thereby effectively serializing those transitions at run-time; for performance reasons, we prefer firing such transitions sequentially over adding an algorithm for run-time conflict resolution.

The hybrid approach may seem similar to *state distribution techniques* in distributed explicit-state model checking. In distributed explicit-state model checking, the state space of the model under verification is distributed over several compute nodes. Each of the nodes subsequently explores its "own" part of the state space. The main challenge is to distribute states over nodes in such a way that communication among nodes is minimal [60]. Several policies for state distribution have been studied. A conceptually straightforward policy is randomly distributing states over nodes [61–64]. Although this policy achieves good load balancing, it also incurs high communication costs. To lower communication costs, alternative distribution policies distribute states based on the property being verified [65] or based on the model under verification [60, 66]. Although related, the difference between distribution of states in the hybrid approach (in terms of medium automata) and distribution of states in distributed model checking is that the states in the hybrid approach are *not* directly part of the full model (i.e., the full product of all medium automata). For instance, if q_1 and q_2 are states of medium automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, then only (q_1, q_2) is a state in the full model $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$. Whereas only q_1 and q_2 are distributed in the hybrid approach, only (q_1, q_2) is available to be distributed in distributed model checking.

8.3. Higher-level concurrent programming

We discuss four major alternatives to protocol dsls for higher-level concurrent programming.

Transactional memory provides a means of controlling concurrent accesses to shared memory as an alternative to lower-level synchronization constructs (e.g., locks or semaphores). Although originally described by Knight in the late 1980s and popularized by Herlihy and Moss in the early 1990s [67,68], the advent of multicore processors caused a renewed interest in

transactional memory from both academia and industry. Support for transactional memory can exist in hardware or in software. Below, we focus on the software variant, first described by Shivat and Touitou [69]. Primarily, transactional memory supports *transactions*: sequences of reads/writes to shared memory that occur atomically. Whenever two running transactions access the same memory location, one of these transactions aborts, rollbacks all the changes it has made so far, and reruns itself. The other transaction may proceed. Whenever a transaction runs to completion without conflicting memory accesses, it commits all the changes it has made. Because a transactional memory system manages transactions transparently to programmers, higher-level transactions should simplify programming compared to lower-level synchronization constructs. One major open issue with transactional memory, however, is performance [70,71]. Another issue is "horizontally" composing individual transactions into full protocol implementations: although every single transaction represents a single protocol, not every single protocol can be represented by a *single* transaction (i.e., generally, the implementation of a protocol may require multiple transactions). As far as we know, no existing transactional memory system supports composition of protocol implementations in a structural way. Lack of such support forces programmers to implement protocols with a mixture of transactions and lower-level computation code, which can be more complex than using higher-level notations and abstractions provided by a DSL for protocols. Note that such horizontal composition is different from "vertical" composition (i.e., nesting) of transactions, which has received considerable attention from the transactional memory community.

Algorithmic skeletons, introduced by Cole in the late 1980s [72], provide software engineers a means of writing programs by composing templates of common patterns of parallel computation and interaction. Algorithmic skeleton APIs conveniently hide all processes and protocols inside their implementation, thereby completely relieving software engineers from the task of implementing protocol specifications. Many algorithmic skeleton APIs exist [73], which seem useful in cases where programs can indeed break down into the algorithmic skeletons provided by those APIs. In other cases, programmers still need to manually implement protocols using lower-level synchronization constructs, which can be more complex than using higher-level notations and abstractions provided by a DSL for protocols. Protocol DSLs are, thus, more generally applicable than algorithmic skeletons. Although not always applicable, the principles behind algorithmic skeletons seem generally useful—also when using a DSL for protocols—as *parallel design patterns* that can help software architects and programmers in specifying and implementing their concurrent programs [74–76], much in the same way as the classical software design patterns help in developing object-oriented programs.

Actors, introduced by Hewitt et al. in the early 1970s [77], constitute another higher-level abstraction for concurrent programming. At run-time, actors mix performing computation with exchanging asynchronous messages in an event-driven fashion. Whenever an actor sends a message to another actor, that message first arrives in the receiving actor's *mailbox*. Once an actor has finished processing a message, it selects a next message from its mailbox, should one exist. While processing messages, actors perform computation and may send messages to other actors. The *pure* actor abstraction hides the underlying shared memory: pure actors communicate with each other only through asynchronous messaging. This asynchronous nature of actors—and their lack of support for synchrony—is their main attraction and a significant weakness at the same time, as synchrony is sometimes essential at the application level. Tasharofi et al., for instance, discovered that programmers often mix Scala/Akka actors [78,79] with Java threads [80], thereby breaking the pure actor abstraction, which can be more complex than using higher-level notations and abstractions provided by a DSL for protocols.

Choreographies recently gained considerable attention from the research community (e.g., [81–84]). A choreography is a global specification of the communication among processes (services, threads, components, etc.), expressed in a high-level language, actually similar to a Reo connector. Unlike our approach of compiling Reo connectors to separate modules of code, however, choreographies are usually *projected* onto processes. This means that with choreographies, there are no separate software entities to enforce protocols among processes; all protocol code is dispersed among process code in process implementations. In early work on choreographies, the per-process communication specifications resulting from projection were primarily used to validate the conformance of (the communication in) process implementations against the choreography. In recent years, projection operations are also used to generate executable code (i.e., process skeletons that, when further filled, are guaranteed to conform to the choreography). Example languages for which such tool support exists include Scribble/Pabble [85,86], Chor [87], and AIOC [88]. A major difference between choreography approaches and our approach is that not every (syntactically) well-formed choreography is amenable to projection (i.e., not every choreography can be implemented), while we can generate code for every well-formed Reo connector. Another difference is that all communication in the same choreography has the same transport characteristics (e.g., asynchronous, reliable, and order-preserving), while Reo allows mixing different transport characteristics within the same Reo connector.

9. Conclusion

Better understanding the differences between centralized-approach compilation and hybrid-approach compilation is crucial to further advance our compilation technology. Initially, we wanted to investigate under which circumstances parallel protocol code (generated by a hybrid-approach compiler), outperforms sequential protocol code (generated by a centralized-approach compiler). Based on our comparison, the answer to this question emerges as:

- Except for cases with overparallelization, hybrid implementations of connectors among more than a few (at least ten to twelve) processes perform at least as good as their centralized counterparts.

Our comparison taught us much more about centralized-/hybrid-approach compilation, though. To summarize our other findings:

- Hybrid-approach compilation may suffer from exponentially sized cas (caused by transition relation explosion) in cases where centralized-approach compilation works fine.
- Centralized-approach compilation may suffer from exponentially sized intermediate products in cases where hybrid-approach compilation works fine.
- Programmers should prefer hierarchically constructed connectors over flat constructed connectors to reduce compilation complexity.
- Hybrid implementations may overparallelize inherently sequential connectors, which leads to poor run-time performance.
- Centralized implementations may in fact have even higher latency than hybrid implementations.
- The more work processes perform, the less important the choice between centralized/hybrid approach becomes (with respect to run-time performance).

These findings suggest that we need a means to alleviate both transition relation explosion (at build-time) and overparallelization (at run-time). To this end, we presented a general framework for moving our hybrid-approach compiler further toward the left of the compilation spectrum in Figs. 2 and 7. Complementarily, in other recent work, we developed an optimization technique to alleviate exponentially sized intermediate products [33]. The development of these two improvements would not have been possible without (the careful analysis of) the experimental results in this paper. It shall be interesting to see how these optimization techniques, in turn, affect our earlier real-world case studies with compiler-generated Reo implementations, such as the NAS Parallel Benchmarks [7] and RSA decryption [89].

We heavily used Reo/connector terminology in this paper as a narrative mechanism. Despite this, we really have been talking about, and investigating, compiler technology for a general kind of communicating automata. Because also other programming languages can have semantics in terms of such automata (e.g., Rebeca [90,91] and Bip [29,30]), our findings may have applications beyond Reo. Moreover, although encountered by us in the context of Reo, mitigating overparallelization seems a generally interesting problem: specifying a system as many parallel processes may feel natural to a system designer, but implementing each of those processes as a thread may give poor performance. By studying overparallelization in terms of automata, our results may advance compilation technology in areas other than Reo, too. For instance, automatically partitioning Bip interaction specifications for generating optimal distributed implementations is still an open problem [59, 92]. Formal relationship between BIP and Reo [30] establishes a basis for extending the impact of our contributions in this paper. Further studies should clarify the extent to which the correspondence between Bip interactions and the automata considered in this paper can be leveraged by reusing our results.

In this paper, we focused on performance in terms of time. We did not consider other efficiency aspects, such as code size and memory. These are, however, important aspects too. In particular, even though compiler-generated implementations of protocols (based on connectors) in some cases can compete with hand-written implementations [7], the size of the generated code is typically much larger than the code programmers would manually write (essentially because transitions are represented explicitly). The same holds for the data structures used at run-time. This can be problematic for systems with limited memory resources in general, and for embedded and cyber–physical systems in particular. In future work, we need to better study these aspects and also develop optimization techniques to address issues related to them.

Finally, it would be interesting to see the extent to which the findings in this paper extend to a timed setting. To study this, we need to extend our compilers from ordinary constraint automata to the existing model of *timed constraint automata* [93]. Compilation of timed constrain automata has, to the best of our knowledge, not been done before and seems a challenging and interesting direction for future work.

Appendix A. Experimental connectors (Fig. 1)

We divided the connector families in Fig. 1 over two categories, except Lock: *k*-producer–single-consumer and single-producer–*k*-consumer. Each category consists of four families. The *k*-producer–single-consumer category contains LateAsyncMerger (Fig. 1g), EarlyAsyncMerger (Fig. 1d), EarlyAsyncBarrierMerger (Fig. 1c), and Alternator (Figs. 1a and 1b); the single-producer–*k*-consumer category contains LateAsyncReplicator (Fig. 1h), EarlyAsyncReplicator (Fig. 1f), LateAsyncRouter (Fig. 1i), and EarlyAsyncOutSequencer (Fig. 1e). In the rest of this appendix, we explain the behavior of these connectors.

A.1. k-producer-single-consumer

With LateAsyncMerger$_k$ (Fig. 1g), whenever producer *i* `writes` a data item on its accessible source node ln$_i$, the connector stores this data item in its only buffer (unless this buffer is already filled by another producer, in which case the `write` suspends until the buffer becomes empty). The relieved producer can immediately continue, possibly before the consumer has completed a `take` for its data item (i.e., communication between a producer and the consumer transpires asynchronously). Whenever the consumer `takes` a data item from its accessible sink node Out, the connector empties the hitherto full buffer. The consumer `takes` data items in the order in which producers `write` them (i.e., communication

between a producer and the consumer transpires undisrupted by other producers). Every round consists of a `write` by a producer and a `take` by the consumer; in every round, two transitions fire.

With EarlyAsyncMerger$_k$ (Fig. 1d), whenever producer i `writes` a data item on its accessible source node In$_i$, the connector stores this data item in its corresponding buffer. The relieved producer can immediately continue, possibly before the consumer has completed a `take` for its data item (i.e., communication between a producer and the consumer transpires asynchronously). Whenever the consumer `takes` a data item from its accessible sink node Out, the connector empties one of the hitherto full buffers, selected nondeterministically. The consumer does not necessarily `take` data items in the order in which producers `write` them (i.e., communication between a producer and the consumer may be interleaved with communication between another producer and the consumer). Every round consists of a `write` by a producer and a `take` by the consumer; in every round, two transitions fire.

Connectors in the EarlyAsyncBarrierMerger family work in largely the same way as those in the EarlyAsyncMerger family, except that the former enforce a barrier on the producers: no producer can `write` its n-th data item until every other producer has completed the `write` of its ($n-1$)-th data item. The consumer may still `take` data items in an order different from the order in which the producers `write` them. Every round consists of a `write` by every producer and k `takes` by the consumer, one for every producer; in every round, $2k$ transitions fire.

With Alternator$_k$ (Figs. 1a and 1b), whenever producer i attempts to `write` a data item on its accessible source node In$_i$, this operation suspends until both (1) the consumer attempts to `take` a data item from its accessible sink node Out, and (2) every other producer j attempts to `write` a data item on its accessible source node In$_j$ (i.e., the producers can `write` only synchronously). Once each of the producers and the consumer attempt to `write`/`take`, the consumer `takes` 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 buffers (i.e., communication between the other producers and the consumer transpires asynchronously). Subsequently, the consumer `takes` the remaining buffered data items in the spatial top-to-bottom order of the producers. Every round consists of a `write` by every producer and k `takes` by the consumer, one for every producer; in every round, k transitions fire.

A.2. Single-producer–k-consumer

With EarlyAsyncReplicator$_k$ (Fig. 1f), whenever the producer `writes` a data item on its accessible source node In, the connector stores this data item in its only buffer. The relieved producer can immediately continue, possibly before the consumers have completed `takes` for its data item (i.e., communication between the producers and a consumer transpires asynchronously). Whenever consumer i attempts to `take` a data item from its accessible sink node Out$_i$, this operation suspends until both (1) the buffer has become full, and (2) every other consumer attempts to `take` a data item (i.e., the consumers can `take` only synchronously). Once the buffer has become full and each of the consumers attempts to `take`, every consumer `takes` a copy of the data item in the buffer, after which the connector empties that buffer. Every round consists of a `write` by the producer and a `take` by every consumer; in every round, two transitions fire.

With LateAsyncReplicator$_k$ (Fig. 1h), whenever the producer `writes` a data item on its accessible source node In, the connector stores a copy of this data item in each of its buffers. The relieved producer can immediately continue, possibly before the consumers have completed `takes` for copies of its data item (i.e., communication between the producers and a consumer transpires asynchronously). Whenever consumer i `takes` a data item from its accessible sink node Out$_i$, the connector empties its corresponding hitherto full buffer. Every round consists of a `write` by the producer and a `take` by every consumer; in every round, $k+1$ transitions fire.

With LateAsyncRouter$_k$ (Fig. 1i), whenever the producer `writes` a data item on its accessible source node In, the connector stores this data item in exactly one of its buffers (instead of a copy in each of its buffers as LateAsyncReplicator$_k$ does), selected nondeterministically. The relieved producer can immediately continue, possibly before the consumer of the selected buffer has completed a `take` for its data item (i.e., communication between the producer and a consumer transpires asynchronously). Whenever consumer i `takes` a data item from its accessible sink node Out$_i$, the connector empties its corresponding full buffer. The consumers do not necessarily `take` data items in the order in which the connector stored those data items in its buffers. Every round consists of a `write` by the producer and a `take` by a consumer; in every round, two transitions fire.

With EarlyAsyncOutSequencer$_k$ (Fig. 1e), whenever the producer `writes` a data item on its accessible source node In, the connector stores this data item in its leftmost buffer. The relieved producer can immediately continue, possibly before a consumer has completed a `take` for its data item (i.e., communication between a producer and the consumers transpires asynchronously). The connector ensures that the consumers can `take` only in their spatial top-to-bottom order. Whenever consumer i `takes` a data item from its accessible sink node Out$_i$, the connector empties its corresponding full buffer. Every round consists of k `writes` by the producer and a `take` by every consumer; in every round, $2k$ transitions fire.

A.3. Lock

Finally, Lock$_k$ represents a classical lock (Fig. 1j). To acquire the lock, process i `writes` an arbitrary data item (i.e., a signal) on its accessible source node Acq$_i$; to release the lock, this process `writes` an arbitrary data item on its accessible

source node Rel$_i$. A `write` on Acq$_i$ suspends until every process j that previously performed a `write` on Acq$_j$ has performed its complementary `write` on Rel$_j$ (i.e., the connector guarantees mutual exclusion). Every round consists of two `write`s by one of the k processes; in every round, two transitions fire.

Appendix B. Experimental results: per-family compilation time charts

Fig. B.15 shows the measured compilation times of compiling the connector families in Fig. 1 for various values of k with `Compiler`$_{centr}$ and `Compiler`$_{hybr}$.
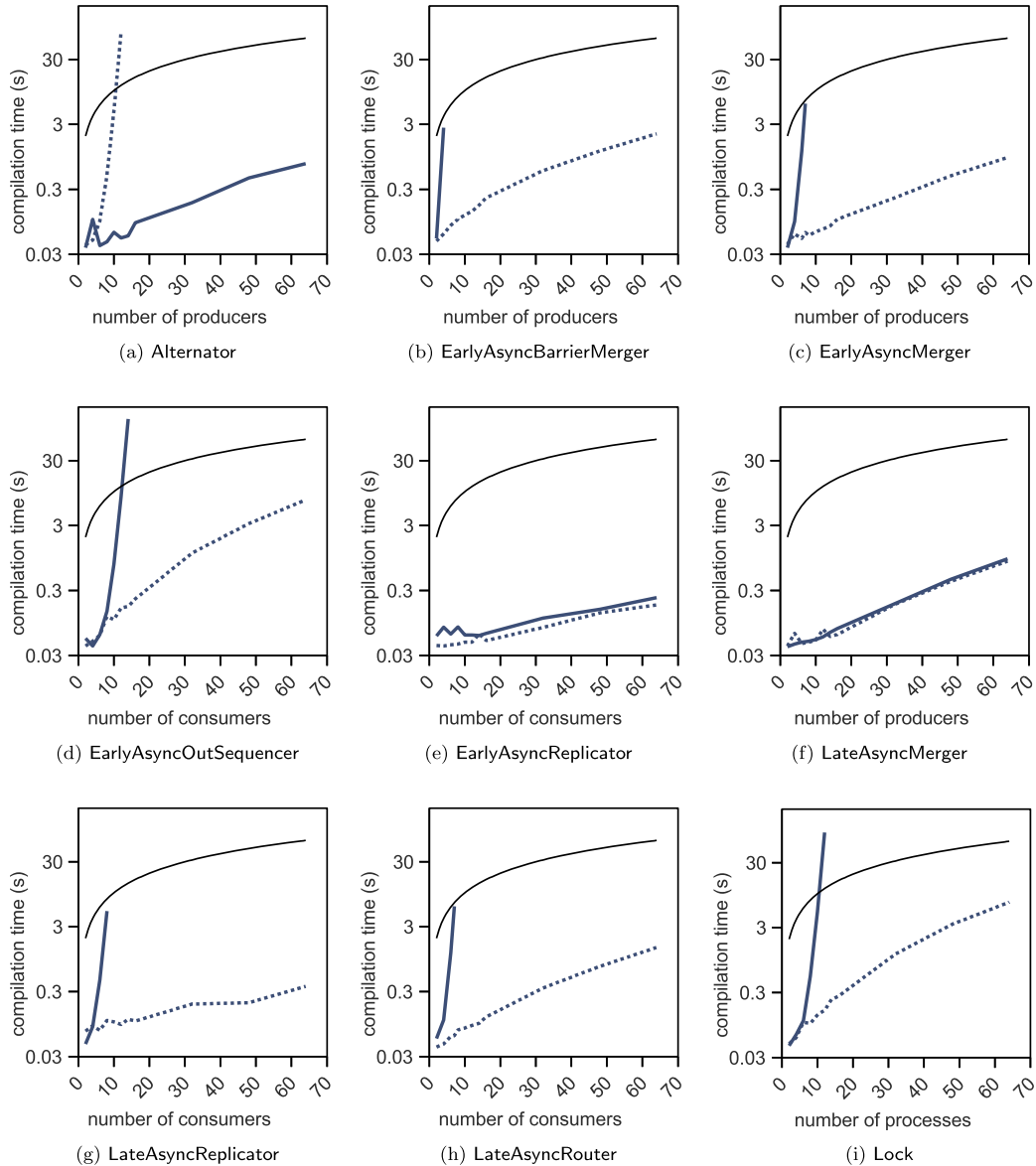

Fig. B.15. Compilation times (solid/dotted lines for centralized/hybrid; thinner black curves for inverse-proportional growth). See Fig. 8 for the same curves plotted in a single chart.

References

[1] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, ACM SIGPLAN Not. 35 (6) (2000) 26–36.

[2] G. Papadopoulos, F. Arbab, Coordination models and languages, Adv. Comput. 46 (1998) 329–400.

[3] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.

[4] F. Arbab, Puff, the magic protocol, in: Formal Modeling: Actors, Open Systems, Biological Systems (Talcott Festschrift), in: LNCS, vol. 7000, Springer, 2011, pp. 169–206.

[5] D. Clarke, D. Costa, F. Arbab, Connector colouring I: synchronisation and context dependency, Sci. Comput. Program. 66 (3) (2007) 205–225.

[6] D. Costa, Formal Models for Component Connectors, Ph.D. thesis, Vrije Universiteit, Amsterdam, 2010.

[7] S.-S. Jongmans, Automata-Theoretic Protocol Programming, Ph.D. thesis, Leiden University, 2016.

[8] S.-S. Jongmans, F. Arbab, Modularizing and specifying protocols among threads, in: Proceedings of PLACES 2012, in: EPTCS, CoRR, vol. 109, 2013, pp. 34–45.

[9] D. Clarke, J. Proença, Partial connector colouring, in: Coordination Models and Languages (Proceedings of COORDINATION 2012), in: LNCS, vol. 7274, Springer, 2012, pp. 59–73.

[10] D. Clarke, J. Proença, A. Lazovik, F. Arbab, Channel-based coordination via constraint satisfaction, Sci. Comput. Program. 76 (8) (2011) 681–710.

[11] J. Proença, D. Clarke, Data abstraction in coordination constraints, in: Advances in Service-Oriented and Cloud Computing (Proceedings of FOCLASA 2013), in: CCIS, vol. 393, Springer, 2013, pp. 159–173.

[12] J. Proença, D. Clarke, Interactive interaction constraints, in: Coordination Models and Languages (Proceedings of COORDINATION 2013), in: LNCS, vol. 7890, Springer, 2013, pp. 211–225.

[13] J. Proença, D. Clarke, E. de Vink, F. Arbab, Decoupled execution of synchronous coordination models via behavioural automata, in: Foundations of Coordination Languages and Software Architectures (Proceedings of FOCLASA 2011), in: EPTCS, CoRR, vol. 58, 2011, pp. 65–79.

[14] J. Proença, D. Clarke, E. de Vink, F. Arbab, Dreams: a framework for distributed synchronous coordination, in: Proceedings of SAC 2012, ACM, 2012, pp. 1510–1515.

[15] J. Proença, Synchronous Coordination of Distributed Components, Ph.D. thesis, Universiteit Leiden, 2011.

[16] S.-S. Jongmans, F. Arbab, Global consensus through local synchronization: a formal basis for partially-distributed coordination, Sci. Comput. Program. 115–116 (2016) 199–224.

[17] S.-S. Jongmans, D. Clarke, J. Proença, A procedure for splitting data-aware processes and its application to coordination, Sci. Comput. Program. 115–116 (2016) 47–78.

[18] S.-S. Jongmans, S. Halle, F. Arbab, Reo: a dataflow inspired language for multicore, in: Proceedings of DFM 2013, IEEE, 2014, pp. 42–50.

[19] S.-S. Jongmans, F. Santini, F. Arbab, Partially-distributed coordination with Reo and constraint automata, Serv. Oriented Comput. Appl. 9 (3) (2015) 311–339.

[20] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in Reo by constraint automata, Sci. Comput. Program. 61 (2) (2006) 75–113.

[21] S.-S. Jongmans, T. Kappé, F. Arbab, Constraint automata with memory cells and their composition, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.03.006, in press.

[22] S.-S. Jongmans, F. Arbab, Can high throughput atone for high latency in compiler-generated protocol code?, in: Fundamentals of Software Engineering (Proceedings of FSEN 2015), in: LNCS, vol. 9392, Springer, 2015, pp. 238–258.

[23] S.-S. Jongmans, F. Arbab, Toward sequentializing overparallelized protocol code, in: Proceedings of ICE 2014, in: EPTCS, CoRR, vol. 166, 2014, pp. 38–44.

[24] F. Arbab, N. Kokash, S. Meng, Towards using Reo for compliance-aware business process modeling, in: Leveraging Applications of Formal Methods, Verification and Validation (Proceedings of ISoLA 2008), in: CCIS, vol. 17, Springer, 2008, pp. 108–123.

[25] B. Changizi, N. Kokash, F. Arbab, A unified toolset for business process model formalization, in: Preproceedings of FESCA 2010, 2010, pp. 147–156.

[26] S. Meng, F. Arbab, C. Baier, Synthesis of Reo circuits from scenario-based interaction specifications, Sci. Comput. Program. 76 (8) (2011) 651–680.

[27] S. Bliudze, J. Sifakis, The algebra of connectors—structuring interaction in BIP, IEEE Trans. Comput. 57 (10) (2008) 1315–1330.

[28] S. Bliudze, J. Sifakis, Causal semantics for the algebra of connectors, Form. Methods Syst. Des. 36 (2) (2010) 167–194.

[29] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: Proceedings of SEFM 2006, IEEE, 2006, pp. 3–12.

[30] K. Dokter, S.-S. Jongmans, F. Arbab, S. Bliudze, Relating BIP and Reo, in: Interaction and Concurrency Experience (Proceedings of ICE 2015), in: EPTCS, CoRR, vol. 189, 2015, pp. 3–20.

[31] F. Arbab, C. Baier, F. de Boer, J. Rutten, M. Sirjani, Synthesis of Reo circuits for implementation of component-connector automata specifications, in: Coordination Models and Languages (Proceedings of COORDINATION 2005), in: LNCS, vol. 3454, Springer, 2005, pp. 236–251.

[32] J.F. Groote, M.R. Mousavi, Basic manipulation of processes, in: Modeling and Analysis of Communicating Systems, The MIT Press, 2014, pp. 141–165, Ch. 9.

[33] S.-S. Jongmans, T. Kappé, F. Arbab, Composing constraint automata, state-by-state, in: Formal Aspects of Component Software (Proceedings of FACS 2015), in: LNCS, vol. 9539, Springer, 2016, pp. 217–236.

[34] S.-S. Jongmans, F. Arbab, Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code?, (Technical Report), Tech. Rep. FM-1503, CWI, 2015.

[35] S.-S. Jongmans, F. Arbab, Overview of thirty semantic formalisms for Reo, Sci. Ann. Comput. Sci. 22 (1) (2012) 201–251.

[36] F. Arbab, J. Rutten, A coinductive calculus of component connectors, in: Recent Trends in Algebraic Development Techniques (Proceedings of WADT 2002), in: LNCS, vol. 2755, Springer, 2003, pp. 34–55.

[37] M.-R. Mousavi, M. Sirjani, F. Arbab, Formal semantics and analysis of component connectors in Reo, in: Foundations of Coordination Languages and Software Architectures (Proceedings of FOCLASA 2005), in: ENTCS, vol. 154, Elsevier, 2006, pp. 83–99.

[38] F. Arbab, R. Bruni, D. Clarke, I. Lanese, U. Montanari, Tiles for Reo, in: Recent Trends in Algebraic Development Techniques (Proceedings of WADT 2008), in: LNCS, vol. 5486, Springer, 2009, pp. 37–55.

[39] S. Meng, F. Arbab, B. Aichernig, L. Aştefănoaei, F.D. Boer, J. Rutten, Connectors as designs: modeling, refinement and test case generation, Sci. Comput. Program. 77 (7–8) (2012) 799–822.

[40] W. Reisig, Petri Nets: An Introduction, EATCS Monographs on Theoretical Computer Science, vol. 4, Springer, 1985.

[41] D. Clarke, Coordination: Reo, Nets, and logic, in: Formal Methods for Components and Objects (Proceedings of FMCO 2007), in: LNCS, vol. 5382, Springer, 2008, pp. 226–256.

[42] R. Bruni, U. Montanari, Zero-safe nets: comparing the collective and individual token approaches, Inf. Comput. 156 (1–2) (2000) 46–89.

[43] K. Dokter, S.-S. Jongmans, F. Arbab, Scheduling games for concurrent systems, in: Coordination Models and Languages (Proceedings of COORDINATION 2016), in: LNCS, vol. 9686, Springer, 2016, pp. 84–100.

[44] M. Sgroi, L. Lavagno, Y. Watanabe, A. Sangiovanni-Vincentelli, Synthesis of embedded software using free-choice Petri nets, in: Proceedings of DAC 1999, ACM, 2009, pp. 805–810.

[45] P.-A. Hsiung, Formal synthesis and code generation of embedded real-time software, in: Proceedings of CODES 2001, ACM, 2001, pp. 208–213.

[46] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, Y. Watanabe, Quasi-static scheduling of independent tasks for reactive systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 24 (10) (2005) 1492–1514.

[47] C. Liu, A. Kondratyev, Y. Watanabe, J. Desel, A. Sangiovanni-Vincentelli, Schedulability analysis of Petri nets based on structural properties, Fundam. Inform. 86 (2008) 325–341.

[48] J. Proença, D. Clarke, Coordination models Orc and Reo compared, in: Foundations of Coordination Languages and Software Architectures (Proceedings of FOCLASA 2007), in: ENTCS, vol. 194, Elsevier, 2008, pp. 57–76.

[49] C. Talcott, M. Sirjani, S. Ren, Comparing three coordination models: Reo, ARC, and PBRD, Sci. Comput. Program. 76 (1) (2011) 3–22.

[50] D. Gelernter, Generative communication in Linda, ACM Trans. Program. Lang. Syst. 7 (1) (1985) 80–112.
[51] M.-D. Feng, W.-F. Wong, C.-K. Yuen, BaLinda lisp: design and implementation, Comput. Lang. 22 (4) (1996) 205–214.
[52] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, T spaces, IBM Syst. J. 37 (3) (1998) 454–474.
[53] M.-D. Feng, Y.-Q. Gao, C.-K. Yuen, Distributed Linda tuplespace algorithms and implementations, in: Parallel Processing: CONPAR 94 – VAPP VI, in: LNCS, vol. 854, Springer, 1994, pp. 581–592.
[54] G. Russello, M. Chaudron, M. van Steen, Dynamically adapting tuple replication for managing availability in a shared data space, in: Coordination Models and Languages (Proceedings of COORDINATION 2005), in: LNCS, vol. 3454, Springer, 2005, pp. 109–124.
[55] R. Bjornson, Linda on Distributed Memory Multiprocessors, Ph.D. thesis, Yale University, 1993.
[56] A. Rowstron, A. Wood, An efficient distributed tuple space implementation for networks of workstations, in: Parallel Processing (Proceedings of Euro-Par 1996), in: LNCS, vol. 1123, Springer, 1996, pp. 510–513.
[57] R. Menezes, R. Tolksdorf, A new approach to scalable Linda-systems based on swarms, in: Proceedings of SAC 2003, ACM, 2003, pp. 375–379.
[58] A. Atkinson, A dynamic, decentralised search algorithm for efficient data retrieval in a distributed tuple space, in: Proceedings of AusPDC 2010, ACM, 2010, pp. 21–30.
[59] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, A framework for automated distributed implementation of component-based models, Distrib. Comput. 25 (5) (2012) 383–409.
[60] S. Orzan, J. van de Pol, M.V. Espada, A state space distribution policy based on abstract interpretation, in: Parallel and Distributed Methods in Verification (Proceedings of PDMC 2004), in: ENTCS, vol. 128, Elsevier, 2005, pp. 35–45.
[61] L. Brim, I. Černá, P. Moravec, J. Šimša, How to order vertices for distributed LTL model-checking based on accepting predecessors, in: Parallel and Distributed Methods in Verification (Proceedings of PDMC 2005), in: ENTCS, vol. 135, Elsevier, 2006, pp. 3–18.
[62] S. Vijzelaar, K. Verstoep, W. Fokkink, H. Bal, Distributed MAP in the SpinJa model checker, in: Parallel and Distributed Methods in Verification (Proceedings of PDMC 2011), in: EPTCS CoRR, vol. 72, 2011, pp. 84–90.
[63] H. Garavel, R. Mateescu, W. Serwe, Large-scale distributed verification using CADP: beyond clusters to grids, in: Parallel and Distributed Methods in Verification (Proceedings of PDMC 2012), in: ENTCS, vol. 296, Elsevier, 2013, pp. 145–161.
[64] J. Barnat, J. Havlíček, P. Ročkai, Distributed LTL model checking with Hash compaction, in: Parallel and Distributed Methods in Verification (Proceedings of PDMC 2012), in: ENTCS, vol. 296, Elsevier, 2013, pp. 79–93.
[65] J. Barnat, L. Brim, I. Černá, Cluster-based LTL model checking of large systems, in: Formal Methods for Components and Objects (Proceedings of FMCO 2005), in: LNCS, vol. 4111, Springer, 2006, pp. 259–279.
[66] E. Khamespanah, M. Sirjani, M. Mousavi, Z.S. Kaviani, M. Razzazi, State distribution policy for distributed model checking of actor models, in: Automated Verification of Critical Systems (Proceedings of AVoCS 2015), in: ECEASST, vol. 72, TU, Berlin, 2015, pp. 1–15.
[67] T. Knight, An architecture for mostly functional languages, in: Proceedings of LFP 1986, ACM, 1986, pp. 105–112.
[68] M. Herlihy, E. Moss, Transactional memory: architectural support for lock-free data structures, ACM SIGARCH Comput. Archit. News (Proc. ISCA 1993) 21 (2) (1993) 289–300.
[69] N. Shavit, D. Touitou, Software transactional memory, Distrib. Comput. 10 (2) (1997) 99–116.
[70] C. Caşcaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transactional memory: why is it only a research toy?, Commun. ACM 51 (11) (2008) 40–46.
[71] M. Herlihy, The Multicore Transformation, Ubiquity 2014, 2014, 1, pp. 1–9.
[72] M. Cole, Algorithmic Skeletons: a Structured Approach to the Management of Parallel Computation, Ph.D. thesis, University of Edinburgh, 1988.
[73] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Softw. Pract. Exp. 40 (12) (2010) 1135–1160.
[74] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape, Commun. ACM 52 (10) (2009) 56–67.
[75] T. Mattson, B. Sanders, B. Massingill, A pattern language for parallel programming, in: Patterns for Parallel Programming, SPS, Addison-Wesley, 2005, pp. 1–6, Ch. 1.
[76] M. McCool, A. Robinson, J. Reinders, Introduction, in: Structured Parallel Programming, Elsevier, 2012, pp. 1–38, Ch. 1.
[77] C. Hewitt, P. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of IJCAI 1973, Morgan-Kauffman, 1973, pp. 235–245.
[78] P. Haller, M. Odersky, Scala actors: unifying thread-based and event-based programming, Theor. Comput. Sci. 410 (2–3) (2009) 202–220.
[79] P. Haller, On the integration of the actor model in mainstream technologies, in: Proceedings of AGERE! 2012, ACM, 2012, pp. 1–6.
[80] S. Tasharofi, P. Dinges, R. Johnson, Why do scala developers mix the actor model with other concurrency models?, in: Object-Oriented Programming (Proceedings of ECOOP 2013), in: LNCS, vol. 7920, Springer, 2013, pp. 302–326.
[81] M. Bravetti, G. Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Software Composition (Proceedings of SC 2007), in: LNCS, vol. 4829, Springer, 2007, pp. 34–50.
[82] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, ACM SIGPLAN Not. (Proc. POPL 2008) 43 (1) (2008) 273–284.
[83] S. Basu, T. Bultan, M. Ouederni, Deciding choreography realizability, ACM SIGPLAN Not. (Proc. POPL 2012) 47 (1) (2012) 191–202.
[84] M. Carbone, K. Honda, N. Yoshida, Structured communication-centered programming for web services, ACM Trans. Program. Lang. Syst. 34 (2) (2012) 8.1–8.78.
[85] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, N. Yoshida, Scribbling interactions with a formal foundation, in: Distributed Computing and Internet Technology (Proceedings of ICDCIT 2011), in: LNCS, vol. 6536, Springer, 2011, pp. 55–75.
[86] N. Ng, J. Coutinho, N. Yoshida, Protocols by default: safe MPI code generation based on session types, in: Compiler Construction (Proceedings of CC 2015), in: LNCS, vol. 9031, Springer, 2015, pp. 212–232.
[87] M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, ACM SIGPLAN Not. (Proc. POPL 2013) 48 (1) (2013) 263–274.
[88] M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, M. Gabbrielli, AIOCJ: a choreographic framework for safe adaptive distributed applications, in: Software Language Engineering (Proceedings of SLE 2014), in: LNCS, vol. 8706, Springer, 2014, pp. 161–170.
[89] M. Krauweel, S.-S. Jongmans, Simpler coordination of JavaScript web workers, in: Proceedings of COORDINATION 2017, in: LNCS, vol. 10319, Springer, 2017, pp. 40–58, http://dx.doi.org/10.1007/978-3-319-59746-1_3.
[90] M. Sirjani, M.-M. Jaghoori, C. Baier, F. Arbab, Compositional semantics of an actor-based language using constraint automata, in: Coordination Models and Languages (Proceedings of COORDINATION 2006), in: LNCS, vol. 4038, Springer, 2006, pp. 281–297.
[91] M. Sirjani, A. Movaghar, A. Shali, F. de Boer, Modeling and verification of reactive systems using Rebeca, Fundam. Inform. 63 (4) (2004) 385–410.
[92] B. Bonakdarpour, M. Bozga, J. Quilbeuf, Model-based implementation of distributed systems with priorities, Des. Autom. Embed. Syst. 17 (2) (2013) 251–276.
[93] F. Arbab, C. Baier, F. de Boer, J. Rutten, Models and temporal logical specifications for timed component connectors, Softw. Syst. Model. 6 (1) (2007) 59–82.