Composing Constraint Automata, State-by-State

Sung-Shik T.Q. Jongmans[1,2,3](✉), Tobias Kappé[4], and Farhad Arbab[3,4]

[1] School of Computer Science, Open University of the Netherlands, Heerlen, The Netherlands
ssj@ou.nl
[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands
[3] Centrum Wiskunde and Informatica, Amsterdam, The Netherlands
[4] Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands

Abstract. The grand composition of n automata may have a number of states/transitions exponential in n. When it does, it seems not unreasonable for the computation of that grand composition to require exponentially many resources (time, space, or both). Conversely, if the grand composition of n automata has a number of states/transitions only linear in n, we may reasonably expect the computation of that grand composition to also require only linearly many resources.

Recently and problematically, we saw cases of linearly-sized grand compositions whose computation required exponentially many resources. We encountered these cases in the context of Reo (a graphical language for coordinating components in component-based software), constraint automata (a general formalism for modeling systems' behavior), and our compiler for Reo based on constraint automata. Combined with earlier research on constraint automata verification, these ingredients facilitate a correctness-by-construction approach to component-based software engineering—one of the hallmarks in Sifakis' "rigorous system design". To achieve that ambitious goal, however, we need to solve the previously stated problem. In this paper we present such a solution.

1 Introduction

Context. Over the past decades, coordination languages emerged for modeling and implementing interaction protocols among components in component-based software. This class of languages includes *Reo* [1,2]. Reo facilitates compositional construction of *connectors*: software entites that embody concurrency protocols for coordinating the synchronization and communication among components. Metaphorically, connectors constitute the "glue" that holds components together in component-based software and mediates their communication. Figure 1 already shows a number of example connectors in their usual graphical syntax. Briefly, a connector consists of a number of *channels* (edges), through which data can flow, and a number of *nodes* (vertices), on which channel ends coincide. The graphical appearance of a channel indicates its *type*; channels of

© Springer International Publishing Switzerland 2016
C. Braga and P.C. Ölveczky (Eds.): FACS 2015, LNCS 9539, pp. 217–236, 2016.
DOI: 10.1007/978-3-319-28934-2_12

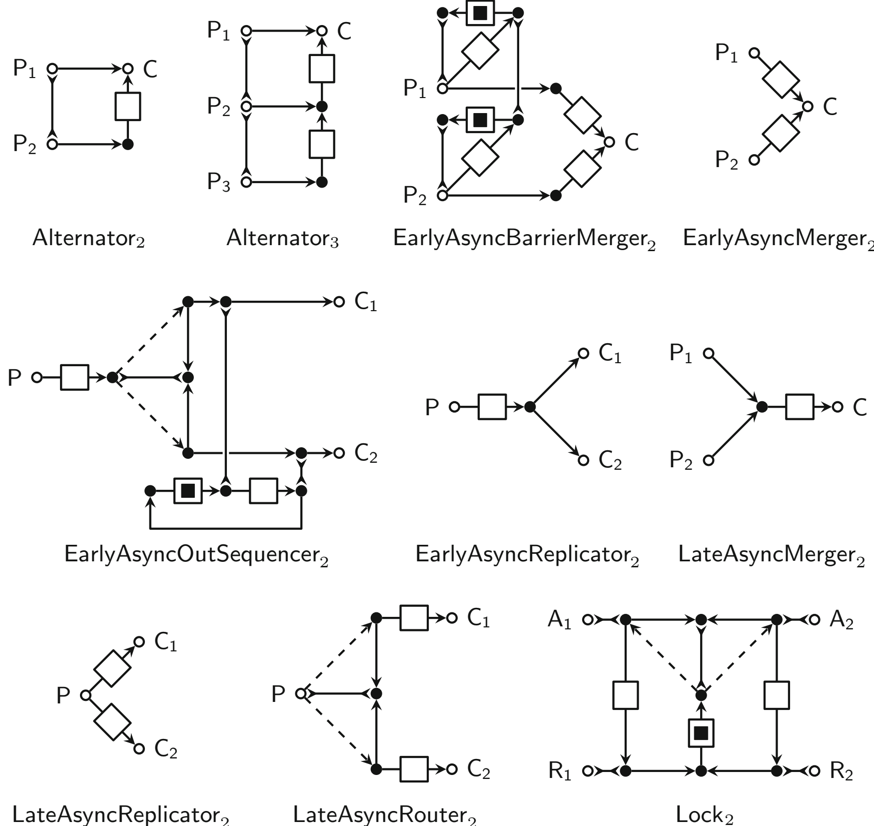

Fig. 1. Example connectors

different types have different data-flow behavior. Figure 1, for instance, includes standard synchronous channels (normal edges) and asynchronous channels with a 1-capacity buffer (rectangle-decorated edges), among others.

Reo has several formal semantics [9], with different purposes. The existence of such formal semantics forms a crucial precondition for Reo-based *rigorous system design* [16]: a design approach proposed by Sifakis centered around the principles of component-based software engineering, separation of concerns, and correctness-by-construction. In this paper, we focus on one particularly important formal semantics of Reo: *constraint automata* (CA) [5]. Constraint automata specify *when* during execution of a connector *which* data flow *where* (i.e., through which channel ends). We can compute the global CA for a connector from the local CAs for that connector's nodes and channels. As such, CAs constitute a *compositional* formal semantics of Reo. Both verification and compilation tools for Reo leverage this compositionality (e.g., [3,4,10,11,13]); the combination of

such tools facilitates a correctness-by-construction approach to component-based software-engineering—one of the hallmarks in Sifakis' rigorous system design.

Problem. Reo's CA-based verification and compilation tools regularly need to compute the grand composition of the local CAs for a connector's *constituents* (i.e., its nodes/channels), to obtain its global CA for subsequent correctness analyses or code generation. The grand composition of n constraint automata, however, may yield a compound CA of a size *exponential* in n. The representation of such exponentially-sized compound CAs may require an exponential amount of space; computation of such CAs may require an exponential amount of time.

Recently, we reported on a number of experiments with our CA-based Reo-to-Java compiler [11]. In these experiments, we indeed observed exponential resource consumption for computing exponentially-sized grand compositions. Curiously, however, we also observed exponential resource consumption for computing *linearly*-sized grand compositions. Whereas exponential resource consumption seems undesirable but understandable for exponentially-sized grand compositions, it seems unacceptable and unintelligible for linearly-sized ones. Before we can achieve the ambitious goal of Reo-based rigorous system design, we must better understand this problem and find a solution.

Contribution. Based on earlier preliminary observations [11], we present a careful analysis of the previously stated problem. Essentially, as we shortly explain in more detail, our existing approach for computing grand compositions sometimes involves the computation of exponentially many "intermediately-reachable-but-finally-unreachable" states in "intermediate compounds", which become unreachable only in the "final compound". Subsequently, we present a solution for this problem in terms of a new approach for computing grand compositions; we prove the corresponding algorithm's correctness using Hoare logic. Finally, we present our implementation of this new approach and evaluate its performance.

In Sect. 2, we discuss preliminaries on Reo and CAs. In Sect. 3, we analyze the previously stated problem. In Sect. 4, we present our solution. In Sect. 5, we evaluate an implementation. Section 7 concludes this paper. An associated technical report contains all formal definitions and in-depth proofs [12].

2 Preliminaries

2.1 Reo

Reo is a graphical language for compositional construction of interaction protocols, manifested as connectors [1,2]. Connectors consist of channels and nodes, organized in a graph-like structure. Every channel consists of two ends and a constraint that relates the timing and the contents of the data-flows at those ends. Channel ends have one of two types: *source ends* accept data into their channels (i.e., a source end of a channel connects to that channel's data source/producer), while *sink ends* dispense data out of their channels (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. Table 1 shows four common channels. Users of Reo may freely extend this set of common channels by defining their own channels with custom semantics.

Table 1. Graphical syntax and informal semantics of common channels

Syntax	*Semantics*
e_1 ⟶ e_2	Synchronously takes a datum d from its source end e_1 and writes d to its sink end e_2.
e_1 ≻—≺ e_2	Synchronously takes data from both its source ends and loses them.
e_1 - - - → e_2	Synchronously takes a datum d from its source end e_1 and nondeterministically either writes d to its sink end e_2 or loses d.
e_1 —□→ e_2 (x)	Asynchronously [takes a datum d from its source end e_1 and stores d in a buffer x], then [writes d to its sink end e_2 and clears x].

Every node has at least one coincident channel end. A node with no coincident sink channel end is called a *source node*. A node with no coincident source channel end is called a *sink node*. A node with both source and sink coincident channel ends is called a *mixed node*. The set of all source nodes and sink nodes of a connector are collectively referred to as its *boundary nodes*. In Fig. 1, we distinguish connectors' white boundary nodes from their shaded mixed nodes.

Every sink channel end coincident on a node serves as a data source for that node. Analogously, every source channel end coincident on a node serves as a data sink for that node. A source node of a connector connects to an output port of a component, which will act as its data source. Similarly, a sink node of a connector connects to an input port of a component, which will act as its data sink. Source nodes permit `put` operations (for components to send data), while sink nodes permit `get` operations (for components to receive data); a connector uses its mixed nodes only for internally routing data.

Contrasting channels, all nodes have the same, fixed data-flow behavior: repeatedly, a node nondeterministically selects an available datum out of one of its data sources and replicates this datum into each of its data sinks. A node's nondeterministic selection and its subsequent replication constitute one atomic execution step; nodes cannot store, generate, or lose data. For a connector to make a global execution step—usually instigated by pending I/O-operations—its channels and its nodes must reach consensus about their combined behavior, to guarantee mutual consistency of their local execution steps (e.g., a node should not replicate a data item into a channel with an already full buffer). Subsequently, connector-wide data-flow emerges. A description of the behavior of the connectors in Fig. 1 appears elsewhere [11].

2.2 Constraint Automata

Although originally developed as a formal semantics of Reo [5], CAs constitute a general operational formalism for modeling the behavior of concurrent systems: every CA models a component, which has a number of *ports* through which it interacts with its environment. Often, we annotate ports with a direction of data-flow (i.e., a component can use a port either for producing data or for consuming data but not for both); in this paper, because these directions do not matter to our current problem, we omit them. To formalize Reo's semantics in terms of CA-based components, we view a channel as a component with two ports (one for each of its two ends), while we view a node with n coincident sink ends and m coincident source ends as a component with $n + m$ ports. Then, we can compositionally compute the CA for a connector by computing the grand composition of the CAs for its constituents. But first, we formally define CAs.

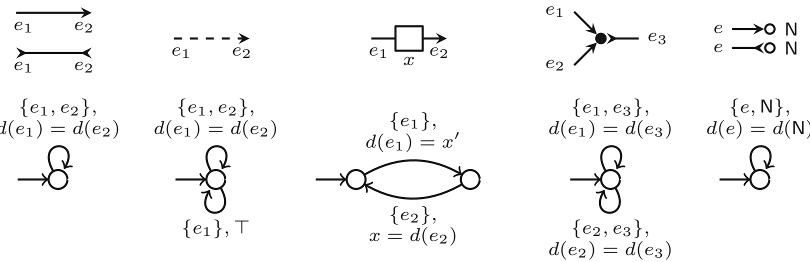

Fig. 2. Constraint automata for the channels in Table 1 (first three from the left), for a mixed node with two incoming and one outgoing channel (fourth from the left), and for two boundary nodes, each with either one incoming or one outgoing channel (fifth from the left). The latter CA is defined not only over the names of its coincident channel ends but also over its own name. (Components use node names—not channel end names—to perform I/O-operations on.)

Structurally, every CA consists of finite sets of states and transitions, which model a component's internal configurations and atomic execution steps. Every transition has a label that consists of two elements: (i) a set, typically denoted by P, containing the names of the ports that have synchronous data-flow in that transition, called a *synchronization constraint*, and (ii) a logical formula, typically denoted by ϕ, that specifies which particular data may flow through which of the ports in P, called a *data constraint*. For instance, the atomic data constraint $d(p_1) = d(p_2)$ means that the same datum flows through ports p_1 and p_2; the atomic data constraint $\top$ means that it does not matter which particular data flow where. Let $\mathbb{D}\text{C}$ denote a universal set of data constraints. More precisely, $\mathbb{D}\text{C}$ serves as the carrier set in some Boolean algebra $(\mathbb{D}\text{C}, \wedge, \vee, \neg, \bot, \top)$, including atoms of the form $d(p_1) = d(p_2)$. The details of data constraints do not matter in this paper, and therefore, we skip them. Let $\mathbb{S}\text{T}$ denote the universal

set of states, let $\mathbb{P}\text{ORT}$ denote the universal set of ports, and let $\mathsf{Dc}(P)$ denote the set of data constraints in which only ports from P occur.

Definition 1. *A constraint automaton is a tuple* $(Q, P^{\text{all}}, \longrightarrow, Q_0)$*, where* $Q \subseteq \mathbb{S}\text{T}$ *is the state space,* $P^{\text{all}} \subseteq \mathbb{P}\text{ORT}$ *is the set of known ports,* $\longrightarrow \subseteq Q \times 2^{P^{\text{all}}} \times \mathsf{Dc}(P^{\text{all}}) \times Q$ *is the transition relation, and* $Q_0 \subseteq Q$ *are the initial states.* $\mathbb{A}\text{UT}$ *is the universal set of constraint automata, ranged over by* α*.*

Figure 2 shows example CAs. Let $\mathsf{St}(\alpha)$, $\mathsf{Port}(\alpha)$, $\mathsf{Tr}(\alpha)$, and $\mathsf{Init}(\alpha)$ denote α's state space, its set of ports, its transition relation, and its initial states.

Our behavioral equivalence in this paper is based on *bisimulation*. We define this equivalence in two steps. First, we define *simulation*.

Definition 2. $\preceq\, \subseteq \mathbb{A}\text{UT} \times \mathbb{A}\text{UT} \times 2^{\mathbb{S}\text{T} \times \mathbb{S}\text{T}}$ *is the relation defined as follows:*

$$\frac{\begin{array}{l}\left[\begin{array}{c}\left[\left[\begin{array}{l}(q_1, P, \phi, q_1') \in \mathsf{Tr}(\alpha_1)\\ \textbf{and}\ \ (q_1, q_2) \in R\end{array}\right] \ \textbf{implies}\ \left[\left[\begin{array}{l}(q_2, P, \phi, q_2') \in \mathsf{Tr}(\alpha_2)\\ \textbf{and}\ \ (q_1', q_2') \in R\end{array}\right] \ \textbf{for some}\ \ q_2'\right]\right]\\ \textbf{for all}\ \ q_1, q_1', q_2, P, \phi\end{array}\right]\\ \textbf{and}\ \ \left[\left[q_1 \in \mathsf{Init}(\alpha_1)\ \ \textbf{implies}\ \left[\begin{array}{c}\left[q_2 \in \mathsf{Init}(\alpha_2)\ \ \textbf{and}\ \ (q_1, q_2) \in R\right]\\ \textbf{for some}\ \ q_2\end{array}\right]\right]\ \textbf{for all}\ \ q_1\right]\\ \textbf{and}\ \ \mathsf{Port}(\alpha_1) = \mathsf{Port}(\alpha_2)\ \ \textbf{and}\ \ R \subseteq \mathsf{St}(\alpha_1) \times \mathsf{St}(\alpha_2)\end{array}}{\alpha_1 \preceq_R \alpha_2}$$

In words, α_2 simulates α_1 under *simulation relation* R—in which case $\alpha_1 \preceq_R \alpha_2$ holds true—whenever we can relate the states of α_1 and α_2 such that: (i) α_2 can mimic every transition that α_1 can make in related states, and (ii) α_2 can already perform such mimicry in any of α_1's initial states. If we care only about the existence of a simulation relation between (the states of) α_1 and α_2 but not about its exact content, we often simply write $\alpha_1 \preceq \alpha_2$. Formally, we "overload" relation symbol $\preceq$ as follows.

Definition 3. $\preceq\, \subseteq \mathbb{A}\text{UT} \times \mathbb{A}\text{UT}$ *is the relation defined as follows:*

$$\frac{\alpha_1 \preceq_R \alpha_2\ \ \textbf{for some}\ \ R}{\alpha_1 \preceq \alpha_2}$$

The definition of bisimulation now straightforwardly follows.

Definition 4. $\simeq\, \subseteq \mathbb{A}\text{UT} \times \mathbb{A}\text{UT} \times 2^{\mathbb{S}\text{T} \times \mathbb{S}\text{T}}$ *is the relation defined as follows:*

$$\frac{\alpha_1 \preceq_R \alpha_2\ \ \textbf{and}\ \ \alpha_2 \preceq_{R^{-1}} \alpha_1}{\alpha_1 \simeq_R \alpha_2}$$

We favor this automata-centric definition of bisimilarity over its definition as the maximal bisimulation on states, because automata are our primary objects of interest instead of their states. As with simulation, if we care only about the existence of a *bisimulation relation* between (the states of) α_1 and α_2 but not about its exact content, we often simply write $\alpha_1 \simeq \alpha_2$ and overload $\simeq$ accordingly.

Definition 5. $\simeq\,\subseteq \mathbb{A}\text{UT} \times \mathbb{A}\text{UT}$ *is the relation defined as follows:*

$$\frac{\alpha_1 \simeq_R \alpha_2 \;\; \textbf{for some} \;\; R}{\alpha_1 \simeq \alpha_2}$$

Note that, as usual with (bi)simulations, $\alpha_1 \simeq \alpha_2$ implies $\left[\alpha_1 \preceq \alpha_2 \text{ and } \alpha_2 \preceq \alpha_1\right]$, but $\left[\alpha_1 \preceq \alpha_2 \text{ and } \alpha_2 \preceq \alpha_1\right]$ does *not* imply $\alpha_1 \simeq \alpha_2$.

To model component composition in terms of CAs, we define the following (synchronous) composition operation.

Definition 6. $\cdot \otimes \cdot : \mathbb{A}\text{UT} \times \mathbb{A}\text{UT} \to \mathbb{A}\text{UT}$ *is the function defined as follows:*

$$\alpha_1 \otimes \alpha_2 = \left(\begin{array}{c} \mathsf{St}(\alpha_1) \times \mathsf{St}(\alpha_2), \mathsf{Port}(\alpha_1) \cup \mathsf{Port}(\alpha_2), \\ \left\{ \begin{pmatrix} (q_1, q_2), \\ P_1 \cup P_2, \\ \phi_1 \wedge \phi_2, \\ (q_1', q_2') \end{pmatrix} \middle| \begin{array}{l} \mathsf{Port}(\alpha_1) \cap P_2 = \mathsf{Port}(\alpha_2) \cap P_1 \\ \textbf{and} \;\; (q_1, P_1, \phi_1, q_1') \in \mathsf{Tr}(\alpha_1) \\ \textbf{and} \;\; (q_2, P_2, \phi_2, q_2') \in \mathsf{Tr}(\alpha_2) \end{array} \right\}, \\ \mathsf{Init}(\alpha_1) \times \mathsf{Init}(\alpha_2) \end{array} \right)$$

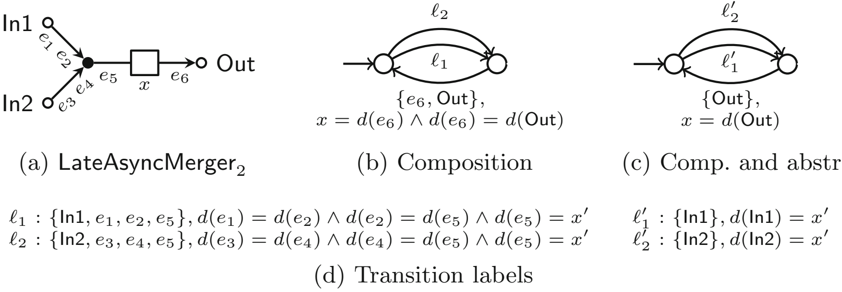

Fig. 3. Composition and abstraction of LateAsyncMerger$_2$ in Fig. 1

Essentially, the previous definition of $\otimes$ formalizes the idea that two components can fire a transition together only if they agree on the involvement of their shared ports. Our composition differs slightly from its original definition [5], where Baier et al. encode the possibility for one CA to *idle*, while the other CA makes a transition, explicitly in the definition of composition. Here, we prefer the equivalent alternative of encoding the idling of components explicitly in their CAs—instead of in the definition of composition—through self-loop transitions labeled with $\emptyset, \top$. This has the advantage of a simpler definition of composition, without losing expressiveness. We stipulate that every example CA that we show has implicit self-loops for idling in each of their states. (In principle, our theory for CAs does not require self-loops; for modeling Reo, however, CAs require self-loops.) Fig. 3 shows an example of composition. We adopt left-associative

notation for $\otimes$ and omit brackets whenever possible (e.g., we write $\alpha_1 \otimes \alpha_2 \otimes \alpha_3$ for $(\alpha_1 \otimes \alpha_2) \otimes \alpha_3$). Similarly, we adopt left-associative notation for pairs of states (e.g., we write (q_1, q_2, q_3) for $((q_1, q_2), q_3)$). Behaviorally, bracketing is insignificant, because $\otimes$ is associative/commutative modulo bisimulation. However, as we reason also structurally about CAs in this paper, bracketing matters.

To compute the formal semantics of a connector, we compute the grand composition of the CAs for its constituents using $\otimes$, in an iterative manner: for an expression $\alpha_1 \otimes \cdots \otimes \alpha_n$, we first compute $\alpha := \alpha_1 \otimes \alpha_2$, then $\alpha := \alpha \otimes \alpha_3$, then $\alpha := \alpha \otimes \alpha_4$, and so on. We call every $\alpha \otimes \alpha_{i<n}$ in this computation an *intermediate compound*; we call $\alpha \otimes \alpha_n$ the *final compound*.

Beside multiplication, Baier et al. defined another operation on constraint automata: *abstraction* [5]. Abstraction removes ports from the observables of a CA, possibly *internalizing* transitions (i.e., making those transitions unobservable from the environment). In practice, abstraction can significantly reduce the size of a CA, both in terms of states and transitions. Although not the main topic of this paper, due to its practical relevance, we use abstraction in Sect. 5. Its formal definition appears below for completeness.

Definition 7. $\cdot \ominus \cdot : \mathbb{A}\text{UT} \times \mathbb{P}\text{ORT} \rightarrow \mathbb{A}\text{UT}$ *is the function defined as follows:*

$$\alpha \ominus p = (\mathsf{St}(\alpha), \mathsf{Port}(\alpha) \backslash \{p\}, \longrightarrow, \mathsf{Init}(\alpha))$$

where $\longrightarrow$ *is the relation defined as follows:*

$$\frac{q_1 \xrightarrow{\emptyset, \phi_1}_{\emptyset} \cdots \xrightarrow{\emptyset, \phi_{n-1}}_{\emptyset} q_n \xrightarrow{P, \phi_n}_{\emptyset} q_{n+1} \ \ \textbf{and} \ \ P \neq \emptyset}{q_1 \xrightarrow{P, \phi_1 \wedge \cdots \wedge \phi_n} q_{n+1}} \qquad \frac{q \xrightarrow{P, \phi}_{\emptyset} q' \ \ \textbf{and} \ \ P \neq \emptyset}{q \xrightarrow{P, \phi} q'}$$

where $\longrightarrow_{\emptyset}$ *is the relation defined as follows:*

$$\frac{(q, P, \phi, q') \in \mathsf{Tr}(\alpha)}{q \xrightarrow{P \backslash \{p\}, \exists p . \phi}_{\emptyset} q'}$$

3 Problem

In ongoing work, we are developing a CA-based Reo-to-Java compiler; in recent work, to study the effectiveness of one of our optimization techniques, we compared the performance of the code generated by our compiler with and without applying that technique [11]. Our comparison featured a number of k-parametric *families* of connectors, where k controls the size of a coordinating connector through its number of coordinated components. Figure 1 shows the $k = 2$ members of the families with which we experimented. One can extend these $k = 2$ members to their $k > 2$ versions in a similar way as how we extended Fig. 1a to b. We selected these families because each of them exhibits different behavior in terms of synchrony, exclusion, nondeterminism, direction, sequentiality, and parallelism, thereby aiming for a balanced comparison.

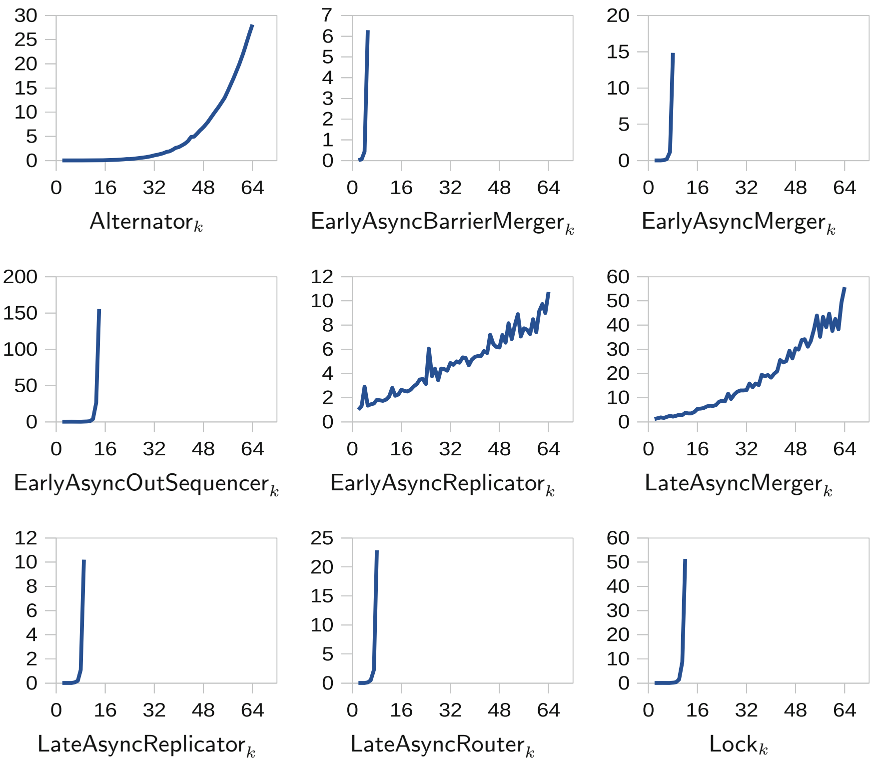

Fig. 4. Computation times (y-axis) for nine k-parametric families, for $2 \leq k \leq 64$ (x-axis). Time is measured in seconds, except for EarlyAsyncReplicator$_k$ and LateAsyncMerger$_k$, where time is measured in milliseconds.

Although we focused our attention primarily on the performance of the generated code, we also made some observations about the performance of our compiler itself. Without applying the optimization technique under investigation, our compiler uses the previously explained *iterative approach* to compute the grand composition of the CAs for a connector's constituents. Figure 4 shows the computation times measured for the k-parametric families under study, for $2 \leq k \leq 64$, averaged over sixteen runs.[1] For six families, the compiler exhausted its available resources (five minutes of time or 2 GB of heap space) long before reaching $k = 64$. The cause: "rapid"—at least exponential—growth in k. For four of these families, we have a good explanation for this phenomenon: the grand compositions computed for EarlyAsyncMerger$_k$, EarlyAsyncBarrierMerger$_k$, LateAsyncReplicator$_k$, and LateAsyncRouter$_k$ grow exponentially in k, such that the amount of resources required to compute those grand compositions logically also grows at least exponentially in k. For the other two families, in contrast,

[1] We recollected the data shown in Fig. 4 specifically for this paper, but we made our initial observations based on our previous data [11].

our measurements seem more difficult to explain: the grand compositions computed for EarlyAsyncOutSequencer$_k$ and Lock$_k$ grow only linearly in k, making an exponential growth in resource requirements rather surprising.

Analysis of the intermediate compounds of EarlyAsyncOutSequencer$_k$ and Lock$_k$ taught us the following: even if final compounds grow linearly in k, their intermediate compounds, as computed by the iterative approach, may nevertheless grow exponentially in k. We can explain this easiest for EarlyAsyncOutSequencer$_k$ (cf. Fig. 1e), through the size of its state space, but the same argument applies to Lock$_k$. 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 datum ■ (i.e., its actual value does not matter); the other buffered channels initially contain nothing. As in the literature [1,2], we call this subconnector Sequencer$_k$. Because no new data can flow into Sequencer$_k$, only ■ cycles through the buffers—ad infinitum—such that only one buffer holds a datum at any time. Consequently, the CA for Sequencer$_k$ has only k states, each of which represents the presence of ■ in exactly one of its k buffers.

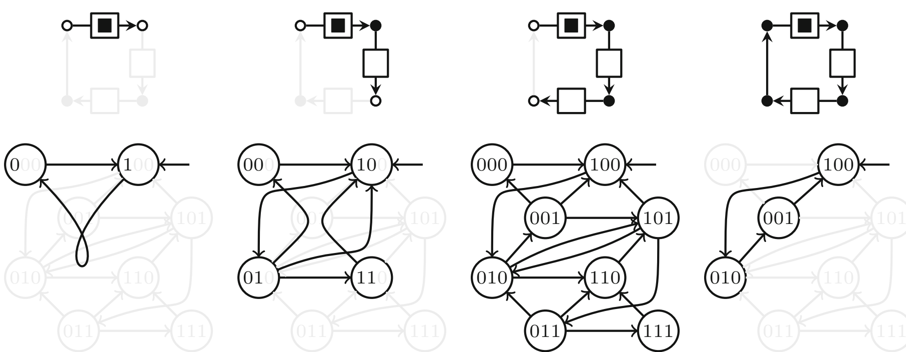

Fig. 5. Grand composition of the CAs for a cycle of three buffered channels (of capacity 1), closed by a synchronous channel. State labels xyz indicate the emptiness/fullness of buffers, where x refers to the first buffer, y to the second buffer, and z to the third buffer; we omitted transition labels to avoid clutter.

However, if we compute the grand composition of the local CAs for Sequencer$_k$'s constituents using the iterative approach, we "close the cycle" only with the very last application of $\otimes$: until then, this soon-to-become-cycle still appears an open-ended chain of buffered channels. Because new data can freely flow into such an open-ended chain, this chain can have a datum in any buffer at any time. Consequently, the CA for the largest chain has 2^k states. Only when we compose this penultimate compound with the last local CA, the state space collapses into k states, as we "find out" that the open-ended chain actually forms a cycle with exactly one datum. Because Sequencer$_k$ constitutes EarlyAsyncOutSequencer$_k$, also EarlyAsyncOutSequencer$_k$ suffers from this problem.

Figure 5 shows our previous analysis in pictures. Most interestingly, the intermediate compounds in Fig. 5 (i.e., the first three automata from the left) contain progressively more states with the following peculiar property: they are reachable from an initial state in those intermediate compounds, called *intermediate-reachability*, but neither those states themselves nor any compound state that they constitute, are reachable in the final compound, called *final-unreachability*. Thus, by using the iterative approach for computing a grand composition, we may spend exponentially many resources on generating a state space that we nearly completely discard in the end. This seems the heart of our problem.

4 Solution

The main idea to solve our problem is to compute grand compositions *state-by-state*, instead of iteratively. In this new approach, we start computing a grand composition from its straightforwardly computable set of initial states. Subsequently, we *expand* each of those states by computing their outgoing compound transitions. These compound transitions enter new compound states, which we subsequently recursively expand. As such, we compute only the reachable states of the final compound, avoiding the unnecessary computation of intermediately-reachable-but-finally-unreachable states. Easy to explain, the main challenge we faced consisted of finding an elegant formalization of this state-by-state approach—including an algorithm—amenable to formal reasoning and proofs. Such proofs are crucially important in the correctness-by-construction principle advocated in rigorous system design for component-based software engineering.

4.1 State-Based Decomposition/Recomposition

We start by formalizing the state-based *decomposition* of a CA into its per-state "subautomata" and the *recomposition* of that CA from those decompositions. Let σ denote the *selection function* (cf. relational algebra) that consumes as input a transition relation $\longrightarrow$ and a state q and produces as output the subrelation of $\longrightarrow$ consisting of precisely the transitions in $\longrightarrow$ that exit q.

Definition 8. $\sigma : 2^{\mathbb{S}\mathrm{T} \times 2^{\mathbb{P}\mathrm{ORT}} \times \mathbb{D}\mathrm{C} \times \mathbb{S}\mathrm{T}} \times \mathbb{S}\mathrm{T} \to 2^{\mathbb{S}\mathrm{T} \times 2^{\mathbb{P}\mathrm{ORT}} \times \mathbb{D}\mathrm{C} \times \mathbb{S}\mathrm{T}}$ *is the function defined as follows:*

$$\sigma_q(\longrightarrow) = \{(q, \hat{P}, \hat{\phi}, \hat{q}') \mid q \xrightarrow{\hat{P}, \hat{\phi}} \hat{q}'\}$$

Next, let $\cdot\langle\cdot\rangle$ denote the *(state-based) decomposition function* that consumes as input an automaton α and a state q and produces as ouput a CA consisting of exactly the same set of states, set of ports, and set of initial states, and with a transition relation consisting of precisely the transitions in α that exit q.

Definition 9. $\cdot\langle\cdot\rangle : \mathbb{A}\mathrm{UT} \times \mathbb{S}\mathrm{T} \to \mathbb{A}\mathrm{UT}$ *is the function defined as follows:*

$$\alpha\langle q\rangle = (\mathsf{St}(\alpha), \mathsf{Port}(\alpha), \sigma_q(\mathsf{Tr}(\alpha)), \mathsf{Init}(\alpha))$$

We call q the *significant state* in $\alpha\langle q\rangle$. The following lemma states that decomposition distributes over composition: instead of first computing the grand composition of n local CAs and then decomposing the resulting global CA with respect to a global state, we can equally first decompose every local CA with respect to its local state and then compute the grand composition of the resulting per-state decompositions. A detailed proof appears in the technical report [12].

Lemma 1. $(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle(q_1, \ldots, q_n)\rangle = \alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle$

The previous definitions (and lemma) cover the essentials of state-based decomposition; in the rest of this subsection, we discuss recomposition. Let $\bigsqcup$ denote a *recomposition function* that consumes as input a set of CAs and produces as output a CA by taking the grand union of the sets of states, sets of ports, sets of transitions, and sets of initial states.

Definition 10. $\bigsqcup \cdot : 2^{\mathbb{A}\text{UT}} \rightarrow \mathbb{A}\text{UT}$ *is the function defined as follows:*

$$\bigsqcup A = \big(\bigcup\{\mathsf{St}(\alpha) \mid \alpha \in A\}, \bigcup\{\mathsf{Port}(\alpha) \mid \alpha \in A\}, \bigcup\{\mathsf{Tr}(\alpha) \mid \alpha \in A\}, \bigcup\{\mathsf{Init}(\alpha) \mid \alpha \in A\}\big)$$

The following lemma states that a CA equals the recomposition of its state-based decompositions. A detailed proof appears in the technical report [12].

Lemma 2. $\alpha = \bigsqcup\{\alpha\langle q\rangle \mid q \in \mathsf{St}(\alpha)\}$

The following theorem states the correctness of the state-by-state approach for grand compositions, as outlined in the beginning of this section. *Roughly*, it states that the grand composition of n local CAs equals the recomposition of that grand composition's state-based decompositions. More precisely, however, it states that this grand composition equals the recomposition of *the composition of state-based decompositions of the local* CA*s*. This is a subtle but important point: it means that to compute the grand composition of n local CAs, we only need to compute compositions of state-based decompositions of those local CAs. We further clarify this point in the next subsection.

Theorem 1

$$\alpha_1 \otimes \cdots \otimes \alpha_n = \bigsqcup\{\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle \mid (q_1, \ldots, q_n) \in \mathsf{St}(\alpha_1) \times \cdots \times \mathsf{St}(\alpha_n)\}$$

Proof (Sketch). By applying Lemma 2, Definition 6 of $\otimes$, and Lemma 1. A detailed proof appears in the technical report [12]. □

4.2 Algorithm

Having formalized de/recomposition, we can now formulate an algorithm for computing the reachable fragment of grand compositions. First, we formalize reachability. We call a state q reachable iff q is an initial state or a finite sequence of k transitions exists that form a path from some initial state to q. Let Reach denote the *reachability function* that consumes as input a CA and produces as output its reachable states.

Definition 11. $\mathsf{Reach} : \mathbb{A}\textsc{ut} \rightarrow 2^{\mathbb{S}\textsc{t}}$ *is the function defined as follows:*

$$\mathsf{Reach}(\alpha) = \mathsf{Init}(\alpha) \cup \left\{ q_k \;\middle|\; \begin{array}{l} (q_1, P_1, \phi_1, q_2), \ldots, (q_{k-1}, P_{k-1}, \phi_{k-1}, q_k) \in \mathsf{Tr}(\alpha) \\ \textbf{and} \;\; q_1 \in \mathsf{Init}(\alpha) \end{array} \right\}$$

Next, let $\lfloor \cdot \rfloor$ denote the *floor function*, which takes as input a CA and produces as output an equivalent—proven below—CA for its reachable states (i.e., the floor function "rounds" a CA "down" to its reachable fragment).

Definition 12. $\lfloor \cdot \rfloor : \mathbb{A}\textsc{ut} \rightarrow \mathbb{A}\textsc{ut}$ *is the function defined as follows:*

$$\lfloor \alpha \rfloor = \bigsqcup \{ \alpha \langle q \rangle \mid q \in \mathsf{Reach}(\alpha) \}$$

The following lemmas state that a CA simulates its floored version and vice versa. Detailed proofs appear in the technical report [12].

Lemma 3. $\alpha \preceq_{\{(q,q) \mid q \in \mathsf{Reach}(\alpha)\}} \lfloor \alpha \rfloor$

Lemma 4. $\lfloor \alpha \rfloor \preceq_{\{(q,q) \mid q \in \mathsf{Reach}(\alpha)\}^{-1}} \alpha$

From these two lemmas, we can immediately conclude the following theorem, which states that a CA and its floored version are bisimulation equivalent.

Theorem 2. $\alpha \simeq_{\{(q,q) \mid q \in \mathsf{Reach}(\alpha)\}} \lfloor \alpha \rfloor$

Proof (Sketch). By applying Lemmas 3 and 4 and Definition 4 of $\simeq$. A detailed proof appears in the technical report [12]. □

$\{\textbf{true}\}$
$A := \emptyset$
$A' := \{ \alpha_1 \langle q_1 \rangle \otimes \cdots \otimes \alpha_n \langle q_n \rangle \mid (q_1, \ldots, q_n) \in \mathsf{Init}(\alpha_1) \times \cdots \times \mathsf{Init}(\alpha_n) \}$
while $\alpha \in A' \setminus A$ **for some** α **do**
  $A := A \cup \{\alpha\}$
  $A' := A' \cup \left\{ \alpha_1 \langle q_1' \rangle \otimes \cdots \otimes \alpha_n \langle q_n' \rangle \;\middle|\; (q, P, \phi, (q_1', \ldots, q_n')) \in \mathsf{Tr}(\alpha) \right\}$
end while
$\{ \bigsqcup A = \lfloor \alpha_1 \otimes \cdots \otimes \alpha_n \rfloor \}$

Fig. 6. Algorithm for computing the grand composition of n autamata using the state-by-state approach

Figure 6 shows an algorithm for computing the grand composition of n local CAs using the state-by-state approach, including a precondition and a postcondition, formulated in terms of de/recomposition and reachability. This algorithm works as described in the beginning of this section. A denotes the subset of so-far computed state-based decompositions whose significant state the algorithm

already has expanded (i.e., the algorithm has processed all CAs in A). A', in contrast, denotes the full set of so-far computed state-based decompositions (i.e., A' contains A such that $A' \setminus A$ contains the CAs that the algorithm still needs to process). After the algorithm terminates, A contains a number of state-based decompositions. The postcondition subsequently asserts that the recomposition of the CAs in A equals the reachable fragment of the grand composition.

$\{\textbf{true}\}$

$\{\mathsf{invar}$
$\quad [A' := \{\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle \mid (q_1, \ldots, q_n) \in \mathsf{Init}(\alpha_1) \times \cdots \times \mathsf{Init}(\alpha_n)\}]$
$\quad [A := \emptyset] \}$

$A := \emptyset$

$A' := \{\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle \mid (q_1, \ldots, q_n) \in \mathsf{Init}(\alpha_1) \times \cdots \times \mathsf{Init}(\alpha_n)\}$

$\{\mathsf{invar}\}$

while $\alpha \in A' \setminus A$ **for some** α **do**

$\quad \{\alpha \in A' \setminus A \textbf{ and } \mathsf{invar} \textbf{ and } |\mathsf{St}(\alpha_1 \otimes \cdots \otimes \alpha_n)| - |A| = z\}$

$\quad \{[\mathsf{invar} \textbf{ and } 0 \leq |\mathsf{St}(\alpha_1 \otimes \cdots \otimes \alpha_n)| - |A| < z]$
$\quad\quad [A' := A' \cup \{\alpha_1\langle q_1'\rangle \otimes \cdots \otimes \alpha_n\langle q_n'\rangle \mid (q, P, \phi, (q_1', \ldots, q_n')) \in \mathsf{Tr}(\alpha)\}]$
$\quad\quad [A := A \cup \{\alpha\}] \}$

$\quad A := A \cup \{\alpha\}$

$\quad A' := A' \cup \{\alpha_1\langle q_1'\rangle \otimes \cdots \otimes \alpha_n\langle q_n'\rangle \mid (q, P, \phi, (q_1', \ldots, q_n')) \in \mathsf{Tr}(\alpha)\}$

$\quad \{\mathsf{invar} \textbf{ and } 0 \leq |\mathsf{St}(\alpha_1 \otimes \cdots \otimes \alpha_n)| - |A| < z\}$

end while

$\{\mathsf{invar} \textbf{ and } [\alpha \notin A' \setminus A \textbf{ for all } \alpha]\}$

$\{\bigsqcup A = \lfloor \alpha_1 \otimes \cdots \otimes \alpha_n \rfloor\}$

Fig. 7. Algorithm for computing the grand composition of n autamata using the state-by-state approach, annotated with assertions for total correctness

Figure 7 shows the algorithm in Fig. 6 annotated with assertions for total correctness; Fig. 8 shows the loop invariant. This invariant consists of four conjuncts. The first conjunct states that $A \cup A'$ contains the initial states in the grand composition. The second conjunct states that the A and A' contain only state-based decompositions of the grand composition. The third conjunct states that every CA in $A \cup A'$ is a state-based decomposition of the grand composition, with respect to some reachable state in that grand composition. The fourth conjunct states that if a CA in A has a transition entering a (global) state q', $A \cup A'$ contains a decomposition of the grand composition with respect to q'. As soon as the loop terminates, the invariant and the negated loop condition imply that every CA in A has a reachable significant state ("soundness"; consequence of the third conjunct) and that, in fact, A contains a CA for every reachable state ("completeness"; consequence of the fourth conjunct).

Theorem 3. *The algorithm in* Fig. 6 *is correct.*

$$\begin{array}{l}
\mathsf{invar}\text{: } \{(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle \mid q \in \mathsf{Init}(\alpha_1 \otimes \cdots \otimes \alpha_n)\} \subseteq A \cup (A' \setminus A) \\
\quad \textbf{and } A, A' \subseteq \{(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle \mid q \in \mathsf{St}(\alpha_1 \otimes \cdots \otimes \alpha_n)\} \\
\quad \textbf{and } \left[\begin{bmatrix} \alpha \in A \cup (A' \setminus A) \textbf{ implies} \\ \left[\begin{bmatrix} \alpha = (\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle \\ \textbf{and } q \in \mathsf{Reach}(\alpha_1 \otimes \cdots \otimes \alpha_n) \end{bmatrix} \textbf{ for some } q\right] \end{bmatrix} \textbf{ for all } \alpha\right] \\
\quad \textbf{and } \left[\begin{bmatrix} [\alpha \in A \textbf{ and } (q, P, \phi, q') \in \mathsf{Tr}(\alpha)] \textbf{ implies} \\ \left[\begin{bmatrix} \alpha' = (\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q'\rangle \\ \textbf{and } \alpha' \in A \cup (A' \setminus A) \end{bmatrix} \textbf{ for some } \alpha'\right] \end{bmatrix} \textbf{ for all } \alpha, q, q', P, \phi\right]
\end{array}$$

Fig. 8. Addendum to Fig. 7

Proof (Sketch). By the assertions in Fig. 7 and the axioms of Hoare logic. A detailed proof appears in the technical report [12].

Note that the invariant refers only to decompositions of the global CA with respect to a global state (e.g., $(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle$ for a global state q), whereas the algorithm refers only to decompositions of local CAs with respect to local states (e.g., $\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle$ for local states $q_1, \ldots, q_n$). Recognizing this difference is important, because it highlights the main advantage of the state-by-state approach: by using only decompositions of local CAs, the algorithm never needs to compute any intermediate compounds, so avoiding a potential source of exponential resource requirements.

5 Implementation, Evaluation, and Discussion

We implemented the state-by-state approach for computing grand compositions as an extension to our CA-based Reo-to-Java compiler. This compiler is implemented in Java and extends the ECT, a collection of plugins for Eclipse that serve as an IDE for Reo (see http://reo.project.cwi.nl).

To evaluate the performance of the state-by-state approach in practice, we experimented with the same k-parameteric families of connectors as those in Fig. 4. Because not only composition but also abstraction play an important role in practice (as mentioned at the end of Sect. 2), we consider three composition–abstraction approaches:

- *Alternating iterative approach.* Variant of the iterative approach where we abstract away all *internal ports* for mixed nodes (which do not contribute to the observable behavior of a connector) in intermediate compounds directly after their computation; this approach alternates between composition and abstraction. It has the advantage that intermediate compounds remain small (i.e., abstraction of internal ports eliminates internal transitions and collapses states together), thereby reducing overall resource consumption (i.e., generally, composing smaller CAs requires fewer resources than composing larger CAs).

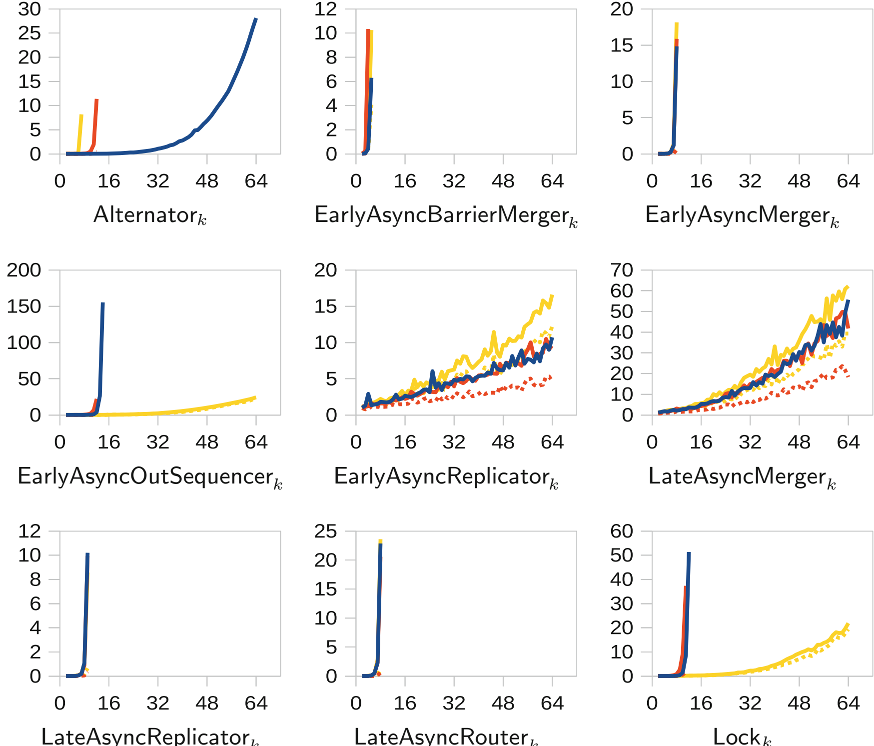

Fig. 9. Computation times (y-axis) for nine k-parametric families, for $2 \leq k \leq 64$ (x-axis), by applying the alternating iterative approach (blue lines), the phased iterative approach before abstraction (dotted-red lines) and after (solid-red lines), and the phased state-by-state approach before abstraction (dotted-yellow lines) and after (solid-yellow lines). Time is measured in seconds, except for EarlyAsyncReplicator$_k$ and LateAsyncMerger$_k$, where time is measured in milliseconds. Page-size versions of these plots appear in the technical report [12] (Color figure online).

– *Phased iterative approach.* Variant of the iterative approach where we abstract away all internal ports only in the final compound and not in intermediate compounds.
– *Phased state-by-state approach.* Variant of the state-by-state approach where we abstract away all internal ports only after the composition phase.

Figure 9 shows the computation times that we measured for the various approaches, connectors, and values of $2 \leq k \leq 64$. We set a timeout of five minutes and bounded the size of the heap at 2 GB.

The four families whose grand compositions grow exponentially in k (i.e., EarlyAsyncBarrierMerger$_k$, EarlyAsyncMerger$_k$, LateAsyncReplicator$_k$, and LateAsyncRouter$_k$) logically provoke exponential growth in resource requirements not only in the iterative approaches (as already observed in Sect. 3) but also in the

phased state-by-state approach. Still, the phased state-by-state approach, performs worse than the alternating iterative approach (at least for $\mathsf{EarlyAsyncBarrierMerger}_k$ and $\mathsf{EarlyAsyncMerger}_k$).

For $\mathsf{EarlyAsyncOutSequencer}_k$ and Lock_k, the phased state-by-state approach has substantially better performance: whereas both the alternating and the phased iterative approaches fail for $k > 14$ (because these approaches require too much resources to successfully complete their computation), the phased state-by-state approach succeeds for all values of k under study. (These two families formed the main motivation for doing the work reported on in this paper.)

For $\mathsf{EarlyAsyncReplicator}_k$ and $\mathsf{LateAsyncMerger}_k$, the phased state-by-state approach seems roughly twice as slow as the iterative approaches. A mundane reason may be that we have not optimized our implementation of the state-by-state approach as aggressively as the iterative approach (which has been under development for several years). Another reason may be that the state-by-state approach is not as cache/memory-friendly as the iterative approach (i.e., locality issues), because the state-by-state approach continuously accesses all local CAs. Moreover—and more seriously—$\mathsf{Alternator}_k$ forms a problematic case for the phased state-by-state approach. Indeed, the alternating iterative approach performs much better, exactly because it abstracts away internal ports as early as possible. Interestingly, early abstraction does not have such a significant effect for all families of connectors under study. This has to do with the particular structure of $\mathsf{Alternator}_k$, explained in detail elsewhere and considered beyond the scope of this paper [10]. Here, the important point is that, although the phased state-by-state approach dramatically improves performance in some cases, it is not a silver bullet. One piece of future work, therefore, concerns the development of heuristics about which composition approach we should apply when. Another piece of future work concerns the investigation of a variant of the state-by-state approach with early abstraction similar to the alternating incremental approach. The main challenge with this is that to perform abstraction, we require certain information that, in the state-by-state approach, seems to become available only once we have completed computing the grand composition. Therefore, we need to develop clever techniques to obtain this kind of information earlier on.

6 Related Work

The main inspiration for our solution in this paper came from Proença's distributed Reo engine [15]. On input of a connector, this engine starts an actor for each of that connector's constituents. Each of these actors has some kind of local automaton (not quite a CA but the differences and details do not matter here) for its corresponding node/channel. Together, the actors run a distributed consensus algorithm to synchronize their behavior, by composing their local behaviors into one consistent global behavior. As part of this consensus algorithm, actors exchange data structures with information about their current state and that state's outgoing transitions (called *frontiers* by Proença). By doing so, the actors effectively compute the composition of their automata at run-time, and only for

their reachable states. Our state-by-state approach for computing grand compositions effectively does a similar computation at compile-time.

Some literature exists on algorithms for computing the composition of CAs. For instance, Ghassemi et al. documented that the order in which a tool composes the CAs in a grand composition matters [7]: although any order yields the same final compound (because composition exhibits associativity and commutativity), different orders may yield diffent intermediate compounds. Some orders may give rise to relatively large intermediate compounds, with high resource requirements as a result, while other orders may keep intermediate compounds small. Choosing the right order, therefore, matters significantly in practice. In the same paper, Ghassemi et al. also briefly mention the idea of computing the composition of *two* CAs in a state-by-state approach, but they do not generalize this to arbitrary grand compositions as we do in this paper. Pourvatan and Rouhy also worked on an algorithm for efficiently computing the composition of two CAs [14]. Their approach consists of a special algebraic representation of CAs, including a reformulation of the composition operation for this representation. Pourvatan and Rouhy claim that their approach computes composition twice as fast as the approach by Ghassemi et al., but evidence remains limited.

State expansion based on reachability also surfaces in what Hopcroft et al. call "lazy evaluation" of subsets in the powerset construction for determinizing a nondeterministic finite automaton in classical automata theory [8]. The fact that we need to compose CAs during the expansion of global states—and explicitly do not want to compute the grand composition beforehand—makes our situation more complex, though. Lemma 1 plays a key role in this respect.

Our work is related also to on-the-fly model checking, proposed by Gerth et al. [6], where the state space under verification is generated as needed during the actual decision procedure instead of in its entirety, beforehand. If a counterexample is found already early during state space generation/exploration, then, no effort gets wasted on precomputing the entire state space. A key difference is our use of Hoare logic to prove our technique's correctness, which to our knowledge has not been done in the context of on-the-fly model checking.

7 Conclusion

Our performance evaluation shows that our new approach for computing grand compositions substantially improves the problematic cases of the existing approach. However, in other cases, our existing approach outperformed our new approach. In future work, we want to investigate heuristics for deciding which of these two approaches we should use when.

Constraint automata comprise a general operational formalism for modeling the behavior of concurrent systems, where every CA models a component. To analyze systems modeled as CAs, efficiently computing the grand composition of those CAs is very important. This makes our work a relevant advancement to the theory and practice of component-based software engineering. In this paper, we focused on the "coordination subsystems"—connectors—among the components.

When expressed in Reo, we can compositionally compute connector behavior in terms of CAs. This enables both verification (e.g., model checking [3,4,13]) and compilation (i.e., code generation [10,11]), whose combination subsequently facilitates a correctness-by-construction approach to component-based software engineering—one of the hallmarks in Sifakis' rigorous system design [16].

We can use our new approach for computing grand compositions also beyond Reo, whenever not only the coordinating connectors' semantics exist as CAs but also the semantics of their coordinated components. For instance, the combination of CAs and Reo has been used to model and verify a simple railway network [3], a biomedical sensor network [4], and an industrial communication platform [13]. To model check temporal logic properties of the composition of the components and connectors of such systems (e.g., the composition never deadlocks), we need to compute the grand composition of the CAs for all components and connectors. Here too, our new approach for computing grand compositions constitutes a valuable alternative to the existing approach. In fact, the abstract approach of computing compound global behavior out of primitive local behavior under a "reachability-based" strategy, to avoid excessive intermediate resource consumption, does not depend on CAs and can be applied also to other models.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. MSCS **14**(3), 329–366 (2004)
2. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011)
3. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 247–267. Springer, Heidelberg (2009)
4. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S., Leister, W.: Design and verification of systems with exogenous coordination using Vereofy. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. SCP **61**(2), 75–113 (2006)
6. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV 1995, pp. 3–18 (1995)
7. Ghassemi, F., Tasharofi, S., Sirjani, M.: Automated mapping of Reo circuits to constraint automata. In: FSEN 2005, ENTCS, vol. 159, pp. 99–115 (2006)
8. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation (2001)
9. Jongmans, S.S., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
10. Jongmans, S.S., Arbab, F.: Toward sequentializing overparallelized protocol code. In: ICE 2014, EPTCS, vol. 166, pp. 38–44 (2014)

11. Jongmans, S.S., Arbab, F.: Can high throughput atone for high latency in compiler-generated protocol code? In: Dastani, M., Sirjani, M. (eds.) FSEN 2015. LNCS, vol. 9392, pp. 238–258. Springer, Heidelberg (2015)
12. Jongmans, S.S., Kappé, T., Arbab, F.: Composing constraint automata, state-by-state (Technical report). Technical report FM-1506, CWI (2015)
13. Klein, J., Klüppelholz, S., Stam, A., Baier, C.: Hierarchical modeling and formal verification. An industrial case study using Reo and Vereofy. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 228–243. Springer, Heidelberg (2011)
14. Pourvatan, B., Rouhy, N.: An alternative algorithm for constraint automata product. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 412–422. Springer, Heidelberg (2007)
15. Proença, J.: Synchronous coordination of distributed components. Ph.D. thesis, Leiden University (2011)
16. Sifakis, J.: Rigorous system design. In: PODC 2014, p. 292 (2014)