Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Constraint automata with memory cells and their composition

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

[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*
[d] *Leiden University, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands*

A R T I C L E   I N F O

Article history:
Received 8 April 2016
Received in revised form 16 March 2017
Accepted 17 March 2017
Available online xxxx

Keywords:
Coordination
Constraint automata
Composition
Reo
Performance

A B S T R A C T

Over the past decades, coordination languages emerged for modeling and implementing concurrency protocols among components in component-based systems. Coordination languages allow programmers to express concurrency protocols at a higher and more appropriate level of abstraction than what traditional programming and scripting languages offer.
In this paper, we extend a significant coordination model—constraint automata—with a mechanism to finitely and compactly deal with infinite data domains, including foundational notions as behavior and equivalence (based on languages), weak and strong congruence (based on bisimulation), and composition. We also address the act of composing a number of simple primitive constraint automata into a complex composite one, by discussing two alternative composition approaches and by analyzing their performance in a number of experiments.

1. Introduction

1.1. Context

Over the past decades, coordination languages have emerged as suitable tools for modeling and implementing concurrency protocols among components in component-based systems. Coordination languages allow programmers to express concurrency protocols at a higher and more appropriate level of abstraction than what traditional programming and scripting languages offer. Practical and efficient formal specification, analysis, and verification of coordination (i.e., higher-level concurrency) protocols require compact representation of infinite behavior and compositional techniques (i.e., those amenable to composition/decomposition). Compositional techniques employ various specific composition operators that combine simpler constructs together to yield more complex constructs. On its own, every such simpler construct can often potentially behave in ways not permitted in composition with its peers in a more complex construct. When applied iteratively to compute a final compound model in multiple composition steps, however, compositional techniques can carry and combine the ultimately disallowed behavior alternatives of the constituents of a complex protocol, producing unnecessarily bloated intermediate models. Recognizing and trimming ultimately disallowed alternatives before intermediate models become bloated can significantly improve the efficiency and applicability of compositional techniques. We present such an approach in this paper.

* Corresponding author.
E-mail addresses: ssj@ou.nl (S.-S.T.Q. Jongmans), kappe@cwi.nl (T. Kappé), farhad@cwi.nl (F. Arbab).

http://dx.doi.org/10.1016/j.scico.2017.03.006
0167-6423/© 2017 Elsevier B.V. All rights reserved.

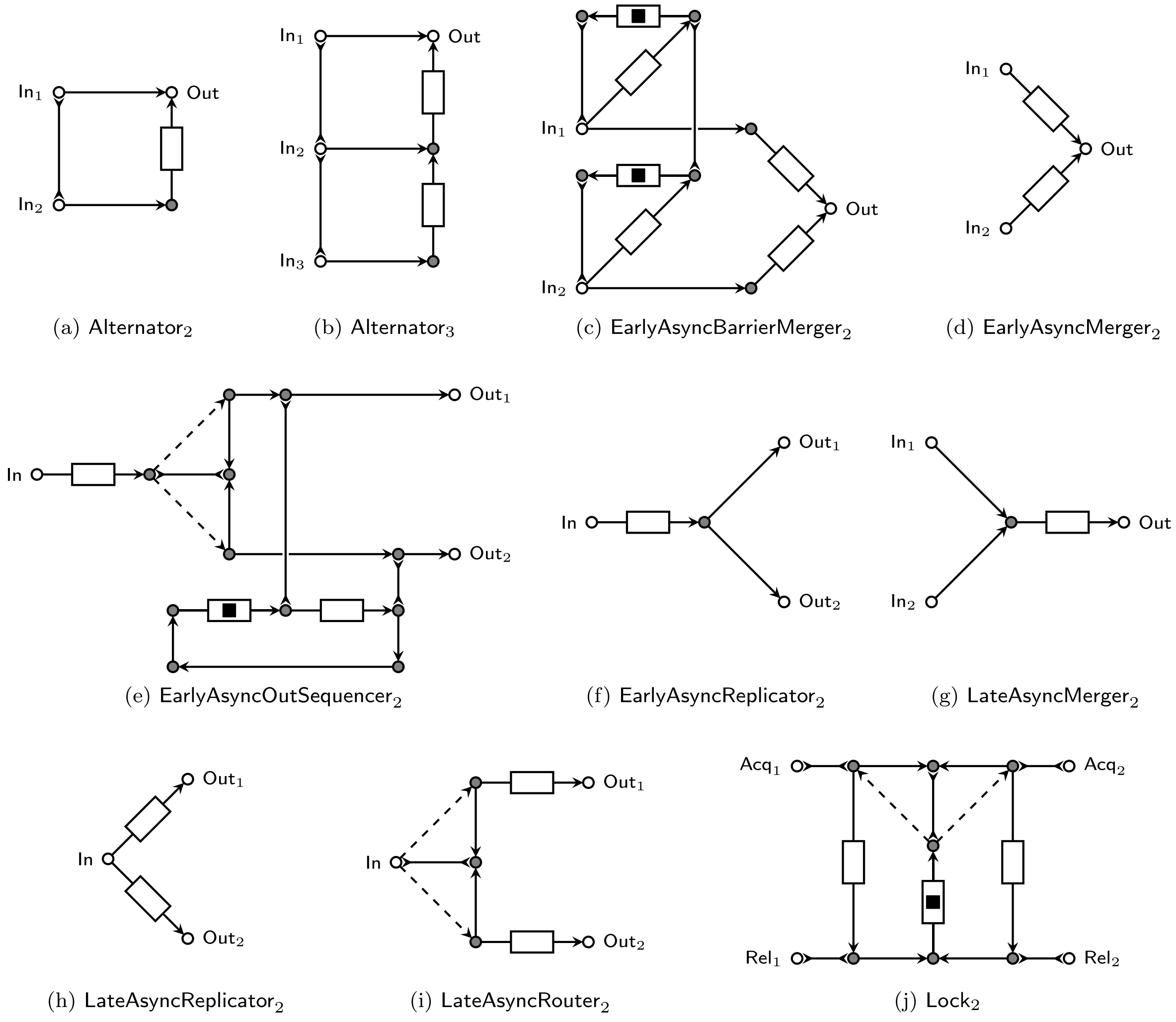


Fig. 1. Example connectors.

A premier example of a coordination language is *Reo* [1,2]. Reo facilitates compositional construction of *connectors*, which embody concurrency protocols for coordinating the synchronization and communication among components. Metaphorically, connectors constitute the "glue" that holds components together in component-based systems and mediates their communication. Fig. 1 shows a number of example connectors in their usual graphical syntax. Briefly, a connector consists of a number of *channels* (edges), through which data *flow*, and a number of *nodes* (vertices), on which channel *ends* coincide. The graphical appearance of a channel indicates its *type*; channels of different types have different data-flow behavior. Fig. 1, for instance, includes standard synchronous channels (normal edges), asynchronous channels with a 1-capacity buffer (rectangle-decorated edges), and two others; Fig. 2 informally describes the data-flow behavior of all channels in Fig. 1. In Fig. 1, white circles represent *boundary nodes* (on which components can perform I/O operations), shaded circles represent *internal nodes* (used only for internal data routing), and a black square inside a rectangle-decorated edge represents that a 1-capacity buffer is full.

In a long-term research project, we are developing Reo compilation technology—including a Reo-to-Java compiler—based on one of Reo's formal semantics [3], namely *constraint automata* [4]. Briefly, a constraint automaton models *when* during execution of a connector *which* data may flow *where* (i.e., through which channel ends). For instance, Fig. 3 shows the constraint automata for the channels in Fig. 2 under a *data domain*—the set of data that may be communicated through a connector—consisting of just 0 and 1, for simplicity. Every transition models an execution step of a connector, characterized by a two-element label: the first element models which channel ends participate in the execution step, while the second element is a logical specification of which data may flow through those participating channel ends. We explain constraint automata in much more detail in the Sections 3, 4, and 5.

On input of (an XML specification of) a connector, our Reo-to-Java compiler works as follows. First, the compiler looks up for every node and for every channel in the input connector a "local" constraint automaton for that particular node or

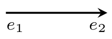
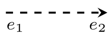
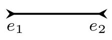
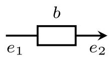
Syntax	*Semantics*
	Accepts a data item d through its channel end e_1 and synchronously offers d through its channel end e_2.
	Either [accepts a data item d through its channel end e_1 and synchronously offers d through its channel end e_2] or [accepts a data item through e_1 and loses it].
	Accepts data items synchronously through both its channel ends and loses them.
	Asynchronously [accepts a data item d through its channel end e_1 and stores d in a buffer b], then [offers d through its channel end e_2 and clears b].

Fig. 2. Graphical syntax and informal semantics of common channels.

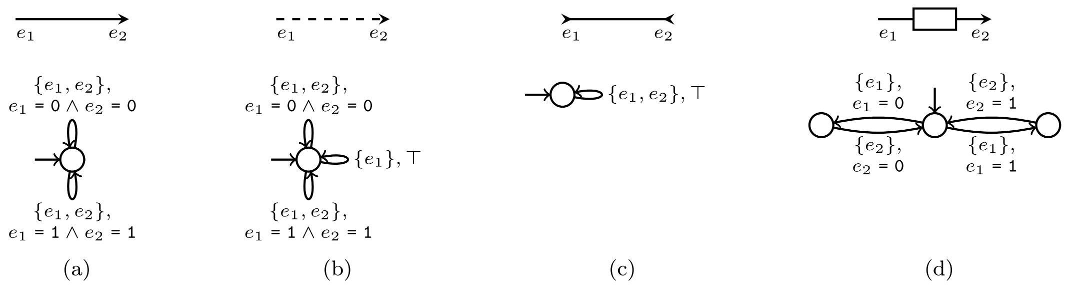


Fig. 3. Constraint automata (without memory cells) for the channels in Fig. 2 under data domain $\{0, 1\}$.

channel. Subsequently, the compiler composes these local constraint automata into a "global" constraint automaton for the whole input connector. Finally, the compiler generates a piece of code for the resulting global constraint automaton, and it automatically weaves this compiler-generated code with hand-written code for components, developed separately and possibly independently of the input connector. The compiler-generated code simulates its corresponding global constraint automaton by firing its transitions as often as possible. In doing so, this code effectively coordinates the system's constituent components at run-time.

The details [5] are substantially more complex, but they are not important for understanding the remainder of this paper. What *is* important to realize, though, is that compiler-generated code has a size proportional to the size of its corresponding global constraint automaton. For instance, if a global constraint automaton has a size exponential in the size of the input connector, the resulting compiler-generated code consists of exponentially many lines of code. Experience suggests that without compensatory measures (i.e., compiler optimization techniques [6,7]), in the worst case, such amounts of code can cause the Java compiler to choke, even on moderately sized connectors (say, consisting of ten channels) [8,5]. For our compilation scheme to be feasible, thus, global constraint automata for input connectors should remain as small as possible; we come back to this point in Section 1.2.

1.2. Problem

As Fig. 3 may already have suggested, the underlying data domain significantly affects a constraint automaton's size. For instance, in the constraint automata in Figs. 3a and 3b, the number of transitions linearly depends on the size of the data domain (i.e., for every particular datum that can flow through a synchronous channel, that channel's constraint automaton has a distinct transition). Even worse, in the constraint automaton in Fig. 3d, not only the number of transitions but also the number of states linearly depends on the size of the data domain (i.e., for every particular datum that the buffer of an asynchronous channel may contain, that channel's constraint automaton has a distinct state). For our Reo-to-Java compiler, whose generated code has a size proportional to the size of its corresponding global constraint automaton, large data domains are thus problematic. And actually, the data domain that our Reo-to-Java compiler needs to support to be practically useful—the set of all Java objects—is not just large but even infinite. This is a serious problem.

One solution is to enhance constraint automata by extending them with mechanisms to better cope with large and infinite data domains. In fact, extending constraint automata with a mechanism to avoid the linear dependence between data domain size and number of transitions (as in the constraint automata in Figs. 3a and 3b) is straightforward. In contrast, however, extending constraint automata with a mechanism to avoid the linear dependence between data domain size and number of states (as in the constraint automaton in Fig. 3d) has not been done before in a systematic way. What we require of such a mechanism is, effectively, a means to explicitly model content of buffers instead of implicitly encoding this as states. A few ad-hoc such extensions exist [9–11], but foundational notions as behavior (e.g., *language*), equivalence (e.g., *language equality*), congruence (e.g., *bisimilarity*), and composition are either still missing or underdeveloped. One practical use case of these theoretical concepts is proving correctness of optimization techniques in our Reo-to-Java compiler. In this use case, we need to show that the behavior of a constraint automaton before and after applying an optimization is equivalent (i.e., behavior-preservation).

1.3. Contribution

With this paper, we make two main contributions. First, we present an extension of constraint automata with a mechanism to finitely and compactly deal with infinite data domains, including definitions of foundational notions as behavior, equivalence, congruence, and composition. Our extension is based on *memory cells*; a rigorous and formal presentation of constraint automata with memory cells (abbreviation: w/mc) as comprehensive as ours in this paper does not yet exist. Second, we consider the issue of efficiently computing the global constraint automaton w/mc for a connector from the local constraint automata w/mc for its nodes and channels: we identify a class of surprising cases where the standard approach to such computation does not work well, propose an alternative algorithm (including a proof of its correctness in Hoare logic), and empirically evaluate this algorithm in experiments. Briefly, what makes the "surprising cases" surprising is that the time to compute the global constraint automaton in these cases is exponential in the size of the connector, whereas the size of the global constraint automaton is only linear; in such cases, one might expect computation time to be linear, too. The "standard approach" [5] to this computation consists of composing local constraint automata one-after-the-other, thereby iteratively forming the global constraint automaton.

Although inspired by our work on Reo, from this point onward, this paper is primarily about constraint automata w/mc. Nevertheless, for completeness, Section 2 starts with preliminaries on Reo, including a detailed description of the connectors in Fig. 1. Readers already familiar with Reo, or readers interested mostly in constraint automata w/mc, can safely skip this section. In Sections 3, 4, and 5, we define the structure of constraint automata w/mc, their behavior, and three operations for their composition. Our first main contribution resides in these three sections. In Sections 6 and 7, we subsequently present two approaches for computing the composition of constraint automata w/mc using the operations defined in Section 5. Our second main contribution resides in these two sections. Section 8 concludes this paper. We discuss related work throughout this paper wherever it is directly relevant to the topic, and mention additional related work in Section 7.3. Proofs of theorems appear in Appendices A–C.

A preliminary version of this paper was previously presented at the 12th International Conference on Formal Aspects of Component Software (FACS 2015) [12]. The new material in this paper consists of a comprehensive presentation of constraint automata with memory cells.

2. Preliminaries on Reo

2.1. Overview

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. Fig. 2 shows four common channels. Users of Reo may freely extend this set of common channels by defining their own channels with custom semantics.

Channel ends coincide on nodes. 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* (or *internal 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 `write` operations (for components to send data), while sink nodes permit `take` 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.

2.2. Examples

Fig. 1 shows a number of example connectors, to further familiarize the reader with Reo. Also, we use these connectors in our experiments in Sections 6 and 7.

First, we explain the behavior of the multiple-producers-single-consumer connectors in Fig. 1. With $\mathsf{LateAsyncMerger}_k$ (Fig. 1g), whenever producer i `write`s a data item on its accessible source node In_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 `take`s a data item from its accessible sink node Out, the connector empties the hitherto full buffer. The consumer `take`s 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 $\mathsf{EarlyAsyncMerger}_k$ (Fig. 1d), whenever producer i `write`s 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 `take`s 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 `take`s by the consumer, one for every producer; in every round, $2k$ transitions fire.

With $\mathsf{Alternator}_k$ (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 `take`s 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 `take`s 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 `take`s by the consumer, one for every producer; in every round, k transitions fire.

Next, we explain the behavior of the single-producer-multiple-consumers connectors in Fig. 1. With $\mathsf{EarlyAsyncReplicator}_k$ (Fig. 1f), whenever the producer `write`s 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 `take`s 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 `take`s 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 $\mathsf{LateAsyncReplicator}_k$ (Fig. 1h), whenever the producer `write`s 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 `take`s for copies of its data item (i.e., communication between the producers and a consumer transpires asynchronously). Whenever consumer i `take`s 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 $\mathsf{LateAsyncRouter}_k$ (Fig. 1i), whenever the producer `write`s 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 $\mathsf{LateAsyncReplicator}_k$ 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 `take`s 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 $\mathsf{EarlyAsyncOutSequencer}_k$ (Fig. 1e), whenever the producer `write`s 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 `take`s a data item from its accessible sink node Out_i, the connector empties its corresponding full buffer. Every round consists of k `write`s by the producer and a `take` by every consumer; in every round, $2k$ transitions fire.

Finally, Lock_k represents a classical lock (Fig. 1j). To acquire the lock, process i `write`s an arbitrary data item (i.e., a signal) on its accessible source node Acq_i; to release the lock, this process `write`s 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.

3. Constraint automata w/mc: structure

3.1. Overview

Although originally developed as a formal semantics of Reo [4], constraint automata constitute a general operational formalism for modeling the behavior of component-based systems. From this perspective, every constraint automaton w/mc models a component with a number of *ports* through which it interacts with its environment and a number of *memory cells* to store data. Every port has a *direction* of data-flow: *input ports* allow a component to receive data, *output ports* allow a component to send data, while *internal ports* allow a component to internally route data between its other ports. If a component has internal ports, we call it a *composite*; otherwise, we call it a *primitive*. To formalize Reo's semantics in terms of constraint automata w/mc, we view a Reo channel with k internal buffers as a component with two ports (one for each of its two ends) and k memory cells, while we view a Reo node with n coincident channel ends as a component with n ports. Subsequently, we can compositionally compute the constraint automaton for a connector by computing the composition of the constraint automata for its nodes and channels.

Instead of first presenting constraint automata as originally defined by Baier et al. [4], we incrementally present our extended version and explain the differences along the way. As our extension is strict, every constraint automaton (without memory cells) is a constraint automaton w/mc (but not vice versa).

Every constraint automaton w/mc consists of a finite set of states, a finite set of transitions, three sets of directed ports, and a set of memory cells. States model the internal configurations of a component; transitions model its atomic execution steps. We use the same composite/primitive terminology for constraint automata w/mc as for components. Different from classical finite automata, 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,[1] 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*. We explain data constraints in detail in Section 3.2

Remark. The presence of memory cells in constraint automata w/mc allows one to explicitly model the content of buffers, whereas the absence of memory cells in constraint automata (without memory cells) forces one to implicitly encode the content of buffers as states (e.g., the constraint automaton in Fig. 3d). The seemingly simple addition of memory cells permeates the entire formalism, however, as we see in the rest of this section and Sections 4 and 5.

3.2. Data constraints

We start by defining some elementary notions.

Definition 1 *(data).* $\mathbb{D}$ denotes the set of all data, ranged over by d.

Definition 2 *(ports).* $\mathbb{P}$ denotes the set of all ports, ranged over by p or e. We use p to range over $\mathbb{P}$ whenever we talk about arbitrary $\underline{\text{p}}$orts, irrespective of what those ports model; we use e to range over $\mathbb{P}$ whenever we talk about ports that specifically model channel $\underline{\text{e}}$nds (i.e., whenever we use constraint automata w/mc specifically to model Reo connectors).

Definition 3 *(memory cells).* $\mathbb{M}$ denotes the set of all memory cells, ranged over by m or b. We use m to range over $\mathbb{M}$ whenever we talk about arbitrary $\underline{\text{m}}$emory cells, irrespective of what those memory cells model; we use b to range over $\mathbb{M}$ whenever we talk about memory cells that specifically model channel $\underline{\text{b}}$uffers.

The exact content of $\mathbb{D}$ depends on the use case. If our goal is compilation to Java, for instance, $\mathbb{D}$ is infinite and contains all Java objects. For verification, in contrast, we usually define $\mathbb{D}$ as a small, finite set of values (e.g., $\{0, 1\}$). In this paper, because we do not concern ourselves with (exhaustive) analysis procedures, the size of $\mathbb{D}$ does not matter: all definitions in this paper work with both finite and infinite, countably or otherwise, $\mathbb{D}$. Henceforth, we write elements of $\mathbb{P}$ in capitalized

[1] With "synchronous data-flow", we mean a set of data exchanges that happen atomically (in the same execution step).

lower case sans-serif (e.g., A, B, C, In_1, Out_2), while we write elements of $\mathbb{D}$ in lower case monospace (e.g., `1`, `3.14`, `true`, `"foo"`). Although data flow through ports always in a certain direction, we do not yet distinguish input ports from output ports; this comes later.

Out of ports and memory cells, we construct *data variables*. Every data variable models a container for data. For instance, ports can hold data, so every port serves as a data variable. Similarly, memory cells can hold data, but the meaning of "to hold" differs in this case. Ports hold data only *during* an execution step (i.e., transiently, in passing). In contrast, memory cells hold data also *before* and *after* an execution step. Consequently, in the context of data variables, a memory cell before an execution step and the same memory cell after that step have a different identity. After all, the content of the memory cell may have changed in between. Therefore—inspired by notation from Petri nets [13]—for every memory cell m, both $^\bullet m$ and $m^\bullet$ serve as data variables: $^\bullet m$ refers to the datum in m before an execution step, while $m^\bullet$ refers to the datum in m after that execution step. We abbreviate sets $\{^\bullet m \mid m \in M\}$ and $\{m^\bullet \mid m \in M\}$ as $^\bullet M$ and $M^\bullet$, where M ranges over sets of memory cells.

Definition 4 *(data variables).* A data variable is an object x generated by the following grammar:

$$x ::= p \mid {}^\bullet m \mid m^\bullet \qquad \text{(data variables)}$$

where p and m range over ports and memory cells. $\mathbb{X}$ denotes the set of all data variables.

We subsequently assign meaning to data variables with *data assignments*.

Definition 5 *(data assignments).* A data assignment is a partial function from data variables to data. $\mathbb{A}\text{SSIGNM} = \mathbb{X} \rightharpoonup \mathbb{D}$ denotes the set of all data assignments, ranged over by σ.

Essentially, a data assignment σ models an execution step involving the ports and memory cells in $\mathrm{Dom}(\sigma)$ and the data in $\mathrm{Img}(\sigma)$. For instance, $\{p_1 \mapsto \mathtt{0}, p_2 \mapsto \mathtt{0}\}$ can model an execution step where datum `0` flows from port p_1 to port p_2, $\{p_1 \mapsto \mathtt{0}, m^\bullet \mapsto \mathtt{0}\}$ can model an execution step where `0` flows from p_1 into memory cell m, while $\{p_2 \mapsto \mathtt{0}, {}^\bullet m \mapsto \mathtt{0}\}$ can model an execution step where `0` flows out of m to p_2. As shown in these examples, data assignments do not capture the *direction* of data-flow: each data-flow is modeled by a data assignment, but every data assignment may model multiple data-flows, depending on directions. As such, data assignments correspond to behavioral observations by a "declarative observer", oblivious to directions, who merely observes "what" happens (data passing through ports) and not "how" this happens (data flowing *from* input ports *to* output ports). Later, in our definition of constraint automata W/MC, we take a more "imperative" perspective by tagging ports as input or output (mainly because this distinction is useful when composing constraint automata W/MC).

Out of data variables, we subsequently construct data constraints. Let M range over subsets of $\mathbb{M}$.

Definition 6 *(data constraints).* A data constraint is an object ϕ generated by the following grammar:

$$\begin{array}{lll} a & ::= \bot \mid \top \mid x = x \mid \mathtt{Keep}(M) & \text{(data atoms)} \\ \ell & ::= a \mid \neg a & \text{(data literals)} \\ \phi & ::= \ell \mid \exists x.\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 & \text{(data constraints)} \end{array}$$

$\mathbb{DC}$ denotes the set of all data constraints.

Henceforth, let $\bigwedge \Phi$ denote the conjunction of the data constraints in Φ, and let $\bigvee \Phi$ similarly denote their disjunction. (This notation is well-defined modulo associativity and commutativity of conjunction and disjunction.)

Every data constraint characterizes a set of data assignments—its se mantics—through an *entailment relation*. Let $\phi[d/x]$ denote data constraint ϕ with datum d substituted for every occurrence of data variable x, and let $\mathsf{Free}(\phi)$ denote its set of *free* data variables.

Definition 7 *(entailment).* $\models \subseteq \mathbb{A}\text{SSIGNM} \times \mathbb{DC}$ denotes the smallest relation induced by the rules in Fig. 4.

Contradiction, tautology, and conjunction have standard semantics [14]. Negation $\neg a$ means that, despite all free variables in a having a value, a does not hold true; the extra condition on the free variables in a ensures the monotonicity of entailment. Data atom $x_1 = x_2$ means that x_1 and x_2 have the same value. Typical examples include $p_1 = p_2$ (possible meaning: a datum flows from port p_1 to port p_2), $p_1 = m^\bullet$ (possible meaning: a datum flows from p_1 into memory cell m), and ${}^\bullet m = p_2$ (possible meaning: a datum flows out of m to p_2); the example data assignments below Definition 5 satisfy the previous three example data atoms. Tautology $\top$ means that it does not matter which data flow through which ports. Predicate $\mathtt{Keep}(M)$ means that the memory cells in M keep their value during an execution step. Henceforth, let $\Rightarrow$ and $\equiv$ denote the implication relation and the equivalence relation on data constraints, derived from $\models$ in the usual way [14].

$$\frac{}{\sigma \models \top} \quad (1)$$

$$\frac{\{b_1, b_2\} \subseteq \mathrm{Dom}(\sigma) \ \textbf{and} \ \sigma(b_1) = \sigma(b_2)}{\sigma \models b_1 = b_2} \quad (2) \qquad \frac{\sigma \models {}^\bullet m = m^\bullet \ \textbf{for all} \ m \in M}{\sigma \models \mathtt{Keep}(M)} \quad (3)$$

$$\frac{\mathsf{Free}(a) \subseteq \mathrm{Dom}(\sigma) \ \textbf{and} \ \sigma \not\models a}{\sigma \models \neg a} \quad (4) \qquad \frac{\sigma \models \phi[d/x] \ \textbf{for some} \ d \in \mathbb{D}}{\sigma \models \exists x.\phi} \quad (5)$$

$$\frac{\sigma \models \phi_1 \ \textbf{or} \ \sigma \models \phi_2}{\sigma \models \phi_1 \vee \phi_2} \quad (6) \qquad \frac{\sigma \models \phi_1 \ \textbf{and} \ \sigma \models \phi_2}{\sigma \models \phi_1 \wedge \phi_2} \quad (7)$$

Fig. 4. Addendum to Definition 7.

Remark. Data constraints in constraint automata (without memory cells) are generated from the following syntax [4]:

$$\phi \ ::= \ \top \mid p = d \mid \phi \vee \phi \mid \neg\phi$$

Notably, in this grammar, equality is asymmetric and existential quantification is missing. Baier et al. subsequently encode $p_1 = p_2$ as $\bigvee\{p_1 = d \wedge p_2 = d \mid d \in \mathbb{D}\}$ and, similarly, $\exists p.\phi$ as $\bigvee\{\phi[d/p] \mid d \in \mathbb{D}\}$. These encodings work only for finite data domains $\mathbb{D}$, though, whereas a symmetric equality as in Definition 6 works also for infinite data domains. Moreover, a symmetric equality enables structural reasoning about data constraints that the asymmetric equality in constraint automata (without memory cells) does not easily support [5]. Finally, $p_1 = p_2$ is easier to automatically check/solve by tools or generated code than $\bigvee\{p_1 = d \wedge p_2 = d \mid d \in \mathbb{D}\}$.

It is straightforward to extend the grammar of data constraints in Definition 6 to a first-order calculus with functions and relations [5]. Such an extension enables modeling a wider range of data-dependent behavior in a compact way. In this paper, because we do not need such behavior, we intentionally keep our grammar simple.

3.3. *Constraint automata w/MC*

We proceed by formally defining a constraint automaton w/MC as a tuple consisting of a set of states Q, a triple of three sets of ports $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}})$, a set of memory cells M, a transition relation $\longrightarrow$, and a set of initial states Q^0. Set P^{all} contains all ports of the component modeled by α, while P^{in} and P^{out} contain only its input ports and its output ports. Although P^{all} contains the union of P^{in} and P^{out}, the converse does not necessarily hold true: as explained in Section 3.1, beside input and output ports, P^{all} may contain also internal ports. Alternatively, one can use an explicit set of internal ports P^{int} instead of P^{all}. However, the definition of our composition operation, shortly, is simpler with P^{all} (i.e., $P_1^{\text{all}} \cup P_2^{\text{all}}$) than with P^{int} (i.e., $P_1^{\text{int}} \cup P_2^{\text{int}} \cup (P_1^{\text{in}} \cap P_2^{\text{out}}) \cup (P_1^{\text{out}} \cap P_2^{\text{in}})$).

Definition 8 *(states).* $\mathbb{Q}$ denotes the set of all states, ranged over by q.

We stipulate that $\mathbb{Q}$ is closed under pairing (i.e., if q_1 and q_2 are states, then (q_1, q_2) is also a state).

Let 2^X denote the power set of some set X.

Definition 9 *(constraint automata w/MC).* A constraint automaton w/MC is a tuple:

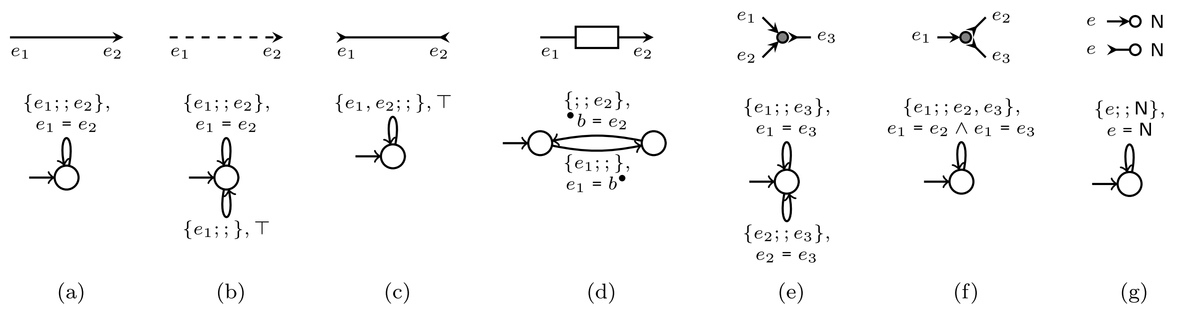
$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, Q^0)$$

where:

- $Q \subseteq \mathbb{Q}$ (states)
- $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) \in 2^{\mathbb{P}} \times 2^{\mathbb{P}} \times 2^{\mathbb{P}}$ such that $\left[P^{\text{in}} \cup P^{\text{out}} \subseteq P^{\text{all}} \ \textbf{and} \ P^{\text{in}} \cap P^{\text{out}} = \emptyset\right]$ (ports)
- $M \subseteq \mathbb{M}$ (memory cells)
- $\longrightarrow \subseteq Q \times 2^{P^{\text{all}}} \times \mathbb{DC} \times Q$ such that: (transitions)

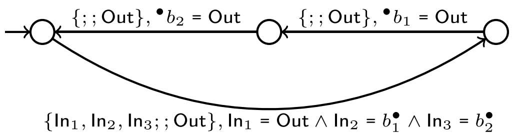
  $$\left[q \xrightarrow{P,\phi} q' \ \textbf{implies} \ \mathsf{Free}(\phi) \subseteq P \cup {}^\bullet M \cup M^\bullet\right] \textbf{for all} \ q, q', P, \phi$$
- $Q^0 \subseteq Q$ (initial states)

$\mathbb{A}$UTOM denotes the set of all constraint automata w/MC, ranged over by α.



e_1 e_2
$\{e_1;;e_2\}$, $e_1 = e_2$
$\{e_1;;e_2\}$, $e_1 = e_2$
$\{e_1;;\}$, $\top$
$\{e_1, e_2;;\}$, $\top$
$\{;;e_2\}$, ${}^\bullet b = e_2$
$\{e_1;;\}$, $e_1 = b^\bullet$
e_1 e_2 e_3
$\{e_1;;e_3\}$, $e_1 = e_3$
$\{e_2;;e_3\}$, $e_2 = e_3$
$\{e_1;;e_2,e_3\}$, $e_1 = e_2 \wedge e_1 = e_3$
e N
$\{e;;\mathsf{N}\}$, $e = \mathsf{N}$
(a)
(b)
(c)
(d)
(e)
(f)
(g)

Fig. 5. Constraint automata w/MC for the channels in Fig. 2 (Figs. 5a, 5b, 5c, and 5d), for a mixed node with two incoming and one outgoing channels (Fig. 5e), for a mixed node with one incoming and two outgoing channels (Fig. 5f), and for two boundary nodes, each with either one incoming or one outgoing channel (Fig. 5g). The latter is defined not only over the names of its coincident channel ends but also over its own name.



$\{;;\mathsf{Out}\}$, ${}^\bullet b_2 = \mathsf{Out}$
$\{;;\mathsf{Out}\}$, ${}^\bullet b_1 = \mathsf{Out}$
$\{\mathsf{In}_1, \mathsf{In}_2, \mathsf{In}_3;;\mathsf{Out}\}$, $\mathsf{In}_1 = \mathsf{Out} \wedge \mathsf{In}_2 = b_1^\bullet \wedge \mathsf{In}_3 = b_2^\bullet$

Fig. 6. Constraint automaton w/MC for $\mathsf{Alternator}_3$ in Fig. 1b.

The requirement $\mathsf{Free}(\varphi) \subseteq P \cup {}^\bullet M \cup M^\bullet$ means that the effect of a transition remains local to its own scope: a transition cannot affect, or be affected by, ports outside its synchronization constraint and memory cells. Alternatively, one can parametrize $\mathbb{DC}$ by ports and memory cells so that $\longrightarrow \subseteq Q \times 2^{P^{\mathsf{all}}} \times \mathbb{DC}(P \cup {}^\bullet M \cup M^\bullet) \times Q$. Baier et al. do this in the original paper on constraint automata [4]. We sacrifice some notational perfection in this respect to avoid notational clutter.

Henceforth, to easily access the individual components of a constraint automaton w/MC α, let $\mathsf{Stat}(\alpha)$ denote α's state space, $\mathsf{Port}(\alpha)$ its set of all ports, $\mathsf{Input}(\alpha)$ its set of input ports, $\mathsf{Outp}(\alpha)$ its set of output ports, $\mathsf{Memor}(\alpha)$ its set of memory cells, $\mathsf{Trans}(\alpha)$ its transition relation, and $\mathsf{Init}(\alpha)$ its set of initial states.

Fig. 5 shows examples of constraint automata w/MC. In this and subsequent figures, we separate input ports, internal ports, and output ports in synchronization constraints with semicolons (in that order). It is instructive to compare the constraint automaton w/MC in Fig. 5d to the constraint automaton (without memory cells) in Fig. 3d: by modeling the content of the buffer of the asynchronous channel with a memory cell, b, we no longer need to encode every possible such content with a distinct state. In other words, whereas the constraint automaton (without memory cells) for this asynchronous channel has as many states (and twice as many transitions) as the size of the data domain plus one, its constraint automaton w/MC has only two states (and two transitions) regardless of the data domain. As another example, Fig. 6 shows the constraint automaton w/MC for $\mathsf{Alternator}_3$ in Fig. 1b. Informally, in the first execution step of $\mathsf{Alternator}_3$, synchronously, a datum flows from node In_1 to node Out, a datum flows from node In_2 into buffer b_1, and a datum flows from node In_3 into buffer b_2. Subsequently, in the second execution step, the datum previously stored in b_1 flows out of that buffer to Out. Finally, in the third execution step, the datum previously stored in b_2 flows out of that buffer to Out. This sequence of three steps repeats itself indefinitely. In Section 5, we demonstrate how to compositionally compute Fig. 6.

In all examples considered so far, and in all those considered in the rest of this paper, unless explicitly stated otherwise, we stipulate that every state q has an *idling transition* $(q, \emptyset, \mathtt{Keep}(\mathsf{Memor}(\alpha)), q)$, without drawing those transitions in figures. An idling transition explicitly models an "execution step" of a component, where no data flow through any of its ports, and where the content of its memory cells does not change. As such, its idling transitions make explicit that a component executes at its own pace and may delay at will, independent of other components. If a constraint automaton w/MC α has an idling transition in each of its states, we call it *idling-enabled*, denoted as $\mathsf{IdlingEnabled}(\alpha)$. Strictly speaking, we do not require idling-enabledness for our definitions to be well-defined (which is why we do not hard-code it in our definitions),[2] but idling-enabledness is a natural assumption at least in the context of Reo (which is why we tacitly assume this property to hold in our examples). Beside more fine-grained models (i.e., increased expressiveness), explicit modeling of idling also simplifies our definition of composition, shortly.

Remark. Already in the original paper on constraint automata (without memory cells) [4], Baier et al. recognize the usefulness of a mechanism similar to memory cells and propose "parametrized constraint automata" as an alternative to "ordinary constraint automata" (without memory cells). Essentially, a parametrized constraint automaton is an ordinary constraint automaton where every state is parametrized with a set of *locations* that may hold a datum in that state (cf. memory cells). Parametrized constraint automata are, however, merely *syntactic sugar* for ordinary constraint automata—"a symbolic repre-

[2] We do need to assume idling-enabledness to prove Theorems 12 and 17, though.

sentation of (non-parameterized) constraint automata" [4]—and can be used only with finite data domains: a parametrized constraint automaton with an infinite data domain would expand to a constraint automaton with infinitely many states or transitions, or to a constraint automaton with a transition labeled with a data constraint with an infinite disjunction. Consequently, with large or infinite data domains, we run into exactly the same problems as those described in Section 1.2. By extending constraint automata with memory cells *as first-class objects*, including their explicit inclusion in definitions of behavior and operations in Sections 4 and 5, we avoid these problems.

Another difference is that idling-enabledness is hard-coded in the definitions of constraint automata (without memory cells), whereas it is optional in our definitions of constraint automata w/MC. This makes our definitions simpler and more expressive: constraint automata w/MC can, for instance, faithfully model lock-step composition of completely independent components, which constraint automata (without memory cells) cannot model. Lock-step composition is, for instance, useful in defining a notion of *soft constraint automata* [15,16], where synchronization constraints and data constraints are tagged with a preference value. Lock-step composition makes it also possible to model the semantics of synchronous languages. One notable candidate is XMAS [17], a graphical hardware description language reminiscent of Reo's graphical notation, but with synchronous semantics.

Remark. By removing also data constraints, constraint automata (without memory cells) further reduce to *port automata*, first studied by Koehler and Clarke [18]. (In turn, the semantic domain of the connector algebras developed by Bliudze and Sifakis essentially consist of single-state port automata [19,20].) Extensions of constraint automata with memory cells include *constraint automata with state memory*, used in work of Pourvatan et al. and formalized in a categorical setting by Krause et al. [9–11]. However, Pourvatan et al. do not define foundational notions as behavior (e.g., language), equivalence (e.g., language equality), and congruence (e.g., bisimilarity), which are crucial for rigorous reasoning about constraint automata w/MC, while Krause et al. define only homomorphy and isomorphy, which is natural in their categorical setting but less so in an automata-theoretic context, where one reasonably may expect languages to play a role, too. Moreover, composition is only partly defined in both the work of Pourvatan et al. and Krause et al.: in terms of the operations presented in Section 5, Pourvatan et al. and Krause et al. define multiplication but not subtraction and aggregation (i.e., join without hide).

4. Constraint automata w/MC: behavior

4.1. Overview

In Section 3, we defined the structure of constraint automata w/MC; in this section, we define their behavior. As with classical finite automata, we associate every constraint automaton w/MC with a *language* consisting of *words* consisting of *letters*. A letter describes an execution step in terms of which data flow through which ports, disregarding memory cells (which are inherently internal and not directly observable by an external observer).

Definition 10 *(letters).* A letter is a partial function from ports to data. $\mathbb{L}\text{ETT} = (\mathbb{P} \rightharpoonup \mathbb{D}) \setminus \emptyset$ denotes the set of all letters, ranged over by λ.

For instance, letter $\{p_1 \mapsto 0, p_2 \mapsto 0\}$ can model an execution step where datum 0 flows from port p_1 to port p_2, letter $\{p_1 \mapsto 0\}$ models an execution step where 0 flows through p_1 (perhaps into a memory cell), while letter $\{p_2 \mapsto 0\}$ models an execution step where 0 flows (perhaps out of a memory cell) through p_2 (cf. the example data assignments below Definition 5).

A word describes an infinite execution.

Definition 11 *(words).* A word is an infinite sequence of letters. $\mathbb{W}\text{ORD} = \mathbb{L}\text{ETT}^{\omega}$ denotes the set of all words, ranged over by w.

For instance, the following three words model executions where (i) the natural numbers synchronously flow from port p_1 to port p_2, (ii) the natural numbers asynchronously flow from p_1 to p_2, and (iii) the natural numbers flow from In_1, In_2, and In_3 to Out according to $\mathsf{Alternator}_3$ in Figs. 1b and 6.

$$\begin{array}{rcl}
\text{(i)} & : & \{p_1 \mapsto 0, p_2 \mapsto 0\}, \{p_1 \mapsto 1, p_2 \mapsto 1\}, \{p_1 \mapsto 2, p_2 \mapsto 2\}, \{p_1 \mapsto 3, p_2 \mapsto 3\}, \{p_1 \mapsto 4, p_2 \mapsto 4\}, \ldots \\
\text{(ii)} & : & \{p_1 \mapsto 0\}, \{p_2 \mapsto 0\}, \{p_1 \mapsto 1\}, \{p_2 \mapsto 1\}, \{p_1 \mapsto 2\}, \{p_2 \mapsto 2\}, \{p_1 \mapsto 3\}, \{p_2 \mapsto 3\}, \ldots \\
\text{(iii)} & : & \{\mathsf{In}_1 \mapsto 0, \mathsf{In}_2 \mapsto 1, \mathsf{In}_3 \mapsto 2, \mathsf{Out} \mapsto 0\}, \{\mathsf{Out} \mapsto 1\}, \{\mathsf{Out} \mapsto 2\}, \\
& & \quad \{\mathsf{In}_1 \mapsto 3, \mathsf{In}_2 \mapsto 4, \mathsf{In}_3 \mapsto 5, \mathsf{Out} \mapsto 3\}, \{\mathsf{Out} \mapsto 4\}, \{\mathsf{Out} \mapsto 5\}, \\
& & \quad\quad \{\mathsf{In}_1 \mapsto 6, \mathsf{In}_2 \mapsto 7, \mathsf{In}_3 \mapsto 8, \mathsf{Out} \mapsto 6\}, \{\mathsf{Out} \mapsto 7\}, \{\mathsf{Out} \mapsto 8\}, \ldots
\end{array}$$

A language describes all possible execution alternatives.

Definition 12 *(languages).* A language is a set of words. $\mathbb{L}\text{ANG} = 2^{\mathbb{W}\text{ORD}}$ denotes the set of all languages.

$$\frac{\begin{array}{l}(q, P, \phi, q') \in \mathsf{Trans}(\alpha)\\ \textbf{and}\ \ \mathrm{Dom}(\mu) = \mathrm{Dom}(\mu') = \mathsf{Memor}(\alpha)\ \ \textbf{and}\ \ \mathrm{Dom}(\lambda) = P\\ \textbf{and}\ \ \lambda \cup \{{}^{\bullet}m \mapsto \mu(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^{\bullet} \mapsto \mu'(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \phi\end{array}}{(q, \lambda w', \mu) \vdash_{\alpha} (q', w', \mu')} \tag{8}$$

$$\frac{\begin{array}{l}(q, \emptyset, \phi, q') \in \mathsf{Trans}(\alpha)\\ \textbf{and}\ \ \mathrm{Dom}(\mu) = \mathrm{Dom}(\mu') = \mathsf{Memor}(\alpha)\\ \textbf{and}\ \ \{{}^{\bullet}m \mapsto \mu(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^{\bullet} \mapsto \mu'(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \phi\end{array}}{(q, w, \mu) \vdash_{\alpha} (q', w, \mu')} \tag{9}$$

Fig. 7. Addendum to Definition 15.

Remark. Our letters, words, and languages go by different names in the literature. For instance, Izadi et al. call letters *records*, words *streams of records*, and languages *languages of records* [21,22]. Alternatively, both Baier et al. [23–26], Klein [27], and Klüppelholz et al. [28–30] call letters *concurrent* I/O *operations* and words I/O *streams*, while Arbab et al. call words *scheduled data streams* [31]. Each of those names refers to the same kind of mathematical object, though. Tuples of *timed data streams* comprise a different but related kind of mathematical object, originally introduced by Rutten and Arbab [32] and later further developed by Arbab into *abstract behavior types* [33]. Every tuple of timed data streams contains one timed data stream for every port of interest. Every timed data stream, in turn, consists of two infinite sequences: a *time stream* of monotonically increasing real numbers and a *data stream* of data. A timed data stream for a port p subsequently models that the i-th datum in the data stream flows through p at the time represented by the i-th real number in the time stream. Consequently, tuples of timed data streams contain not only information about the order in which data-flow through ports takes place but also more precise timing information.

4.2. Behavior

The memory cells in constraint automata w/MC may remind one of *stacks* in pushdown automata: both memory cells and stacks register behaviorally relevant—yet externally unobservable—information. In defining the runs of a constraint automaton w/MC, we therefore adopt terminology and notation from pushdown automata, as follows. An *instantaneous description* (q, w, μ) of a constraint automaton w/MC consists of three elements: its current state q, the remaining word w ("input tape"), and the current content of memory cells μ ("stack"), called a *snapshot*.

Definition 13 *(snapshots).* A snapshot is a partial function from memory cells to data. $\mathbb{S}\text{NAPSH} = \mathbb{M} \rightharpoonup \mathbb{D}$ denotes the set of all memory snapshots, ranged over by μ.

Definition 14 *(instantaneous descriptions).* $\mathbb{D}\text{ESCR} = \mathbb{Q} \times \mathbb{W}\text{ORD} \times \mathbb{S}\text{NAPSH}$ denotes the set of all instantaneous descriptions.

A constraint automaton w/MC can *move* from one instantaneous description to the next by firing a transition out of its current state, thereby possibly changing its state, consuming the first letter of the remaining word, and changing the content of memory cells. More precisely, for a constraint automaton w/MC α with memory cells M to move from instantaneous description $(q, \lambda w, \mu)$ to instantaneous description (q', w, μ'), several conditions must hold. First, α should have a transition (q, P, ϕ, q') from state q to state q'. Second, snapshots μ and μ' should have exactly M as their domain (i.e., in making a transition, α cannot affect, or be affected by, memory cells that it does not know about). Third, letter λ should satisfy the synchronization constraint of the transition: λ should have exactly P as its domain. Finally, the data assignment composed of λ, μ and μ' should satisfy data constraint ϕ.

Importantly, unless ϕ explicitly states otherwise (e.g., by using the `Keep` predicate), the content of memory cells in M can arbitrarily change during a move. In other words, we do not hard-code in the semantics of a constraint automaton w/MC that a move should preserve the content of memory cells; if the content *should* be preserved, this must be explicitly stated in the data constraint. This is not always necessary, though. For instance, if a memory cell is to be read from only once after a write, the constraint automaton w/MC should preserve the content of this memory cell only between the write and the first subsequent read; after the read, it does not matter what happens to the content of the memory cell (until the next write), and we allow the automaton to change it however it likes. This approach is in line with the nondeterministic nature of constraint automata w/MC and significantly simplifies the definition of moves (compared to the deterministic alternative where memory cells always keep their content unless explicitly stated otherwise).

As an alternative to the previous conditions, α can also move from (q, w, μ) to (q', w, μ') if a transition from q to q' with an empty synchronization constraint exists: such an *internal transition* does not directly contribute to α's observable behavior.

Definition 15 *(move).* $\vdash \subseteq \mathbb{D}\text{ESCR} \times \mathbb{D}\text{ESCR} \times \mathbb{A}\text{UTOM}$ denotes the smallest relation induced by the rules in Fig. 7.

Let α denote a constraint automaton w/MC, let $q_1 \in \mathsf{Init}(\alpha)$ denote an initial state (existentially quantified), let w_1 denote a word, and let $\mu_1 : \mathsf{Memor}(\alpha) \to \mathbb{D}$ denote an initial snapshot (also existentially quantified). If α has an infinite run (i.e., infinite sequence of successive moves) starting from instantaneous description (q_1, w_1, μ_1), word w_1 belongs to the language of α. In that case, α *accepts* w_1. Because we use constraint automata w/MC only to represent languages—branching transition structures do not directly matter to us—we call the language of α its behavior. Henceforth, as a notational convention, let all variables that are not bound in set-builder notation be existentially quantified.

Definition 16 *(behavior).* $\mathsf{Behav} : \mathbb{A}\text{UTOM} \to \mathbb{L}\text{ANG}$ denotes the function defined by the following equation:

$$\mathsf{Behav}(\alpha) = \{w_1 \mid q_1 \in \mathsf{Init}(\alpha) \textbf{ and } (q_1, w_1, \mu_1) \vdash_\alpha (q_2, w_2, \mu_2) \vdash_\alpha \cdots\}$$

In this definition, $(q_1, w_1, \mu_1) \vdash_\alpha (q_2, w_2, \mu_2) \vdash_\alpha \cdots$ means that there exist an infinite sequence of states $q_1, q_2, \ldots$, an infinite sequence of words $w_1, w_2, \ldots$, and an infinite sequence of snapshots $\mu_1, \mu_2, \ldots$, such that for all $i > 0$, we have $(q_i, w_i, \mu_i) \vdash_\alpha (q_{i+1}, w_{i+1}, \mu_{i+1})$.

Definition 16 is insensitive to *divergence* (i.e., infinitely many internal steps): $\mathsf{Behav}(\alpha)$ contains only the words that induce infinite sequences of steps such that always eventually an observable step happens. It is possible to make the definition sensitive to divergence by omitting (9) in Fig. 7 and by redefining $\mathbb{L}\text{ETT}$ in Definition 10 as $\mathbb{P} \rightharpoonup \mathbb{D}$ (thereby making $\emptyset$ a valid letter). Because we are interested only in observable data-flows, however, we do not pursue divergence-sensitivity in this paper.[3]

4.3. Equivalence and congruence

Two automata are *behaviorally equivalent* if they have the same behavior.

Definition 17 *(behavioral equivalence).* $\approx\, \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ denotes the smallest relation induced by the following rule:

$$\frac{\mathsf{Behav}(\alpha_1) = \mathsf{Behav}(\alpha_2)}{\alpha_1 \approx \alpha_2} \tag{10}$$

Theorem 1. *$\approx$ is an equivalence*

As usual, $\approx$ is an equivalence but *not* a congruence for the operations presented in Section 5 (witness: the constraint automata versions of processes $a(b+c)$ and $(ab)+(ac)$ in process algebra, which are equivalent but incongruent for parallel composition). To remedy this situation, again as usual, we define a congruence on constraint automata w/MC that implies $\approx$, based on bisimulation, taking into account the branching structure of constraint automata w/MC. Only languages and $\approx$ truly matter to us in the end, though; congruence, and the branching structure of constraint automata w/MC, serve just as a means to simplify reasoning about languages and $\approx$. We define congruence in two steps. First, we define *simulation* and *behavioral preorder*.

Definition 18 *(simulation).* $\preceq\, \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \times 2^{\mathbb{Q}\times\mathbb{Q}}$ is the relation defined as follows:

$$\frac{\left[\left[\left[\begin{array}{l} q_1 \xrightarrow{P,\phi_1} q_1' \\ \textbf{and } q_1 \; R \; q_2 \end{array}\right] \textbf{ implies } \phi_1 \Rightarrow \bigvee \left\{ \phi_2 \;\middle|\; \begin{array}{l} q_2 \xrightarrow{P,\phi_2} q_2' \\ \textbf{and } q_1' \; R \; q_2' \end{array} \right\}\right] \textbf{ for all } q_1, q_1', q_2, P, \phi_1\right] \\ \textbf{and } R \subseteq Q_1 \times Q_2 \textbf{ and } Q_1^0 \subseteq \{q_1 \mid q_1 \; R \; q_2 \textbf{ and } q_2 \in Q_2^0\}}{(Q_1, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow_1, Q_1^0) \preceq_R (Q_2, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow_2, Q_2^0)} \tag{11}$$

In words, α_2 simulates α_1 under R—in which case $\alpha_1 \preceq_R \alpha_2$ holds true—whenever three conditions hold. The first condition in Definition 18 states that if states q_1 and q_2 are related by R, then α_2 in q_2 must be able to "mimic" every transition of α_1 in q_1. A transition labeled by P, ϕ_1 is mimicked by a transition labeled by P, ϕ_2 whenever (i) the target states of the two transitions are related by R and (ii) ϕ_1 implies ϕ_2. Requirement (ii) means that any letter (i.e., data-flow) that satisfies ϕ_1 also satisfies ϕ_2 (i.e., any data-flow induced by the simulated transition can, indeed, be simulated by the simulating transition). Requirement (i) means that α_2 in q_2 can mimic α_1 in q_1 not only for the next transition, but also for all future transitions to come. The second condition in Definition 18 states that R is a relation on the states of α_1 and α_2. The third condition in Definition 18 states that every initial state of α_1 must be simulated by an initial state of α_2. 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.

[3] Our behavioral congruence (Definition 20) *is* sensitive to divergence, though. Because we usually (including in this paper) reason about behavior modulo behavioral congruence (or stronger) instead of behavioral equivalence, thus, our results are also applicable in a divergence-sensitive setting.

Definition 19 *(behavioral preorder).* $\preceq \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ is the relation defined as follows:

$$\frac{\alpha_1 \preceq_R \alpha_2 \textbf{ for some } R}{\alpha_1 \preceq \alpha_2} \tag{12}$$

Theorem 2. *$\preceq$ is a preorder*

The definitions of *bisimulation* and *behavioral congruence* now straightforwardly follow.

Definition 20 *(bisimulation).* $\simeq \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \times 2^{\mathbb{Q}\times\mathbb{Q}}$ is the relation defined as follows:

$$\frac{\alpha_1 \preceq_R \alpha_2 \textbf{ and } \alpha_2 \preceq_{R^{-1}} \alpha_1}{\alpha_1 \simeq_R \alpha_2} \tag{13}$$

Note that we need to take the inverse of R for the simulation between α_2 and α_1 (second conjunct in the antecedent of the rule). A variant where α_2 is allowed to simulate α_1 under R_1, and α_1 is allowed to simulate α_2 under R_2, with $R_2 \neq R_1^{-1}$, also exists (simulation equivalence), but is weaker than bisimulation.

Definition 21 *(behavioral congruence).* $\simeq \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ is the relation defined as follows:

$$\frac{\alpha_1 \simeq_R \alpha_2 \textbf{ for some } R}{\alpha_1 \simeq \alpha_2} \tag{14}$$

Theorem 3. *$\simeq$ is an equivalence*

In Section 5, we show that $\simeq$ is not just an equivalence but actually a congruence for composition. Note also that $\alpha_1 \simeq \alpha_2$ 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 $\alpha_1 \simeq \alpha_2$.

The following theorem states that behavioral congruence implies behavioral equivalence, as desired.

Theorem 4. $\simeq \subseteq \approx$

5. Constraint automata w/MC: operations

5.1. Overview

As explained in Section 1.1, composition—the act of constructing complex constraint automata w/MC out of simpler ones—plays a leading role in our Reo compiler. Having defined the structure and behavior of constraint automata w/MC in Sections 3 and 4, in this section, we define a number of operations—*multiplication*, *subtraction*, and *aggregation*—on constraint automata w/MC that facilitate their compositional construction. We use multiplication to "join" constraint automata w/MC on their shared ports, thereby constructing larger constraint automata w/MC; we use subtraction and aggregation to "hide" information from constraint automata w/MC, notably internal ports and internal transitions. To compose constraint automata w/MC, we first multiply them, subsequently subtract the internal ports from the resulting product automaton, and finally aggregate internal transitions in the resulting difference automaton. We exemplify this process in this section for the Alternator_3 connector.

5.2. Multiplication

Multiplication consumes two constraint automata w/MC α_1 and α_2 as input and produces a constraint automaton w/MC as output. We formally define multiplication on constraint automata w/MC as a partial function. This partiality represents that not all constraint automata w/MC can compose into a new one: two constraint automata w/MC can compose only if (i) each of their *shared ports* serves as an input port in one constraint automaton w/MC and as an output port in the other and (ii) these two constraint automata w/MC have no shared memory cells. Thus, constraint automata w/MC in composition cannot share anything except ports, which naturally corresponds to the intuition that memory cells are private.

Because constraint automata w/MC have a rather involved structure, the formal definition of multiplication may look deceivingly complex. Therefore, we first present a more informal, operational description to explain the main concepts involved. Let $\alpha_1 = (Q_1, (P_1^{\text{all}}, P_1^{\text{in}}, P_1^{\text{out}}), M_1, \longrightarrow_1, Q_1^0)$, and let $\alpha_2 = (Q_2, (P_2^{\text{all}}, P_2^{\text{in}}, P_2^{\text{out}}), M_2, \longrightarrow_2, Q_2^0)$. Assuming that conditions (i) and (ii) hold true, we take the following steps to multiply α_1 and α_2.

- We take the Cartesian product of Q_1 and Q_2 as the new set of states. Similarly, we take the Cartesian product of Q_1^0 and Q_2^0 as the new set of initial states.

- We take the union of P_1^{all} and P_2^{all} as the new set of all ports. Subsequently, we put every port in $P_1^{\text{in}} \cup P_2^{\text{in}}$ in the new set of input ports, except those that serve *also* as output port in the other automaton. Similarly, we put every port in $P_1^{\text{out}} \cup P_2^{\text{out}}$ in the new set of output ports, except those that serve *also* as input port in the other automaton. Ports with complementary directions (e.g., those that serve as input port in α_1 and as output port in α_2) become internal ports in the product.
- We take the union of M_1 and M_2 as the new set of memory cells. (It is not necessary to take the disjoint union, because we assume that M_1 and M_2 are already disjoint.)
- Finally, we must construct a new transition relation out of $\longrightarrow_1$ and $\longrightarrow_2$. The intuition here is as follows: α_i ($i \in \{1,2\}$) can fire a transition in the product of α_1 and α_2 only whenever α_j ($j \in \{1,2\} \setminus \{i\}$) can synchronously fire a transition *involving the same shared ports*. In other words, α_1 and α_2 must always agree on data-flow through their shared ports.
  One notable case of applying this rule is the case where α_1 and α_2 fire transitions that do not involve any shared ports; in that case, these automata "vacuously" agree. A notable subcase within this notable case is the case where α_j fires an *internal transition*, which does not involve ports whatsoever (i.e., the synchronization constraint of an internal transition is empty; firing such a transition can only change the content of memory cells). This happens, for instance, whenever α_j fires an idling transition.

Definition 22 *(multiplication)*. $\otimes : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ denotes the partial function defined by the following equation:

$$\left(\begin{array}{c} Q_1, \\ \begin{pmatrix} P_1^{\text{all}}, \\ P_1^{\text{in}}, \\ P_1^{\text{out}} \end{pmatrix}, \\ M_1, \\ \longrightarrow_1, \\ Q_1^0 \end{array}\right) \otimes \left(\begin{array}{c} Q_2, \\ \begin{pmatrix} P_2^{\text{all}}, \\ P_2^{\text{in}}, \\ P_2^{\text{out}} \end{pmatrix}, \\ M_2, \\ \longrightarrow_2, \\ Q_2^0 \end{array}\right) = \left(\begin{array}{c} Q_1 \times Q_2, \\ \begin{pmatrix} P_1^{\text{all}} \cup P_2^{\text{all}}, \\ (P_1^{\text{in}} \setminus P_2^{\text{out}}) \cup (P_2^{\text{in}} \setminus P_1^{\text{out}}), \\ (P_1^{\text{out}} \setminus P_2^{\text{in}}) \cup (P_2^{\text{out}} \setminus P_1^{\text{in}}) \end{pmatrix}, \\ M_1 \cup M_2, \\ \longrightarrow_\otimes, \\ Q_1^0 \times Q_2^0 \end{array}\right), \quad \textbf{if} \left[\begin{array}{l} P_1^{\text{all}} \cap P_2^{\text{all}} = \\ \quad (P_1^{\text{in}} \cap P_2^{\text{out}}) \cup (P_1^{\text{out}} \cap P_2^{\text{in}}) \\ \textbf{and } M_1 \cap M_2 = \emptyset \end{array}\right]$$

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

$$\frac{q_1 \xrightarrow{P_1, \phi_1}_1 q_1' \textbf{ and } q_2 \xrightarrow{P_2, \phi_2}_2 q_2' \textbf{ and } P_1^{\text{all}} \cap P_2 = P_2^{\text{all}} \cap P_1}{q \xrightarrow{P_1 \cup P_2, \phi_1 \wedge \phi_2}_\otimes q'} \tag{15}$$

Theorem 5. $\alpha_1 \otimes \alpha_2 \simeq \alpha_2 \otimes \alpha_1$

Theorem 6. $\alpha_1 \otimes (\alpha_2 \otimes \alpha_3) \simeq (\alpha_1 \otimes \alpha_2) \otimes \alpha_3$

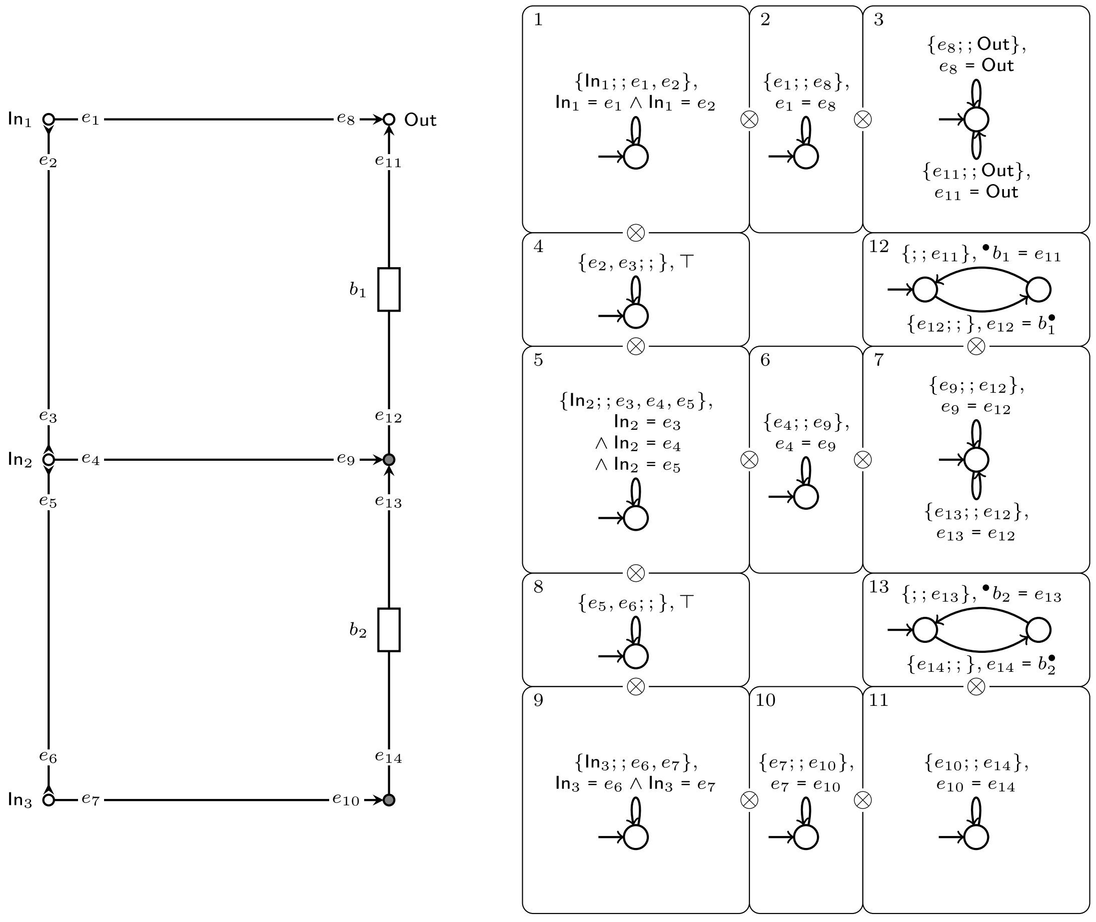
We use left-associative notation for $\otimes$ and omit brackets whenever possible (e.g., we write $\alpha_1 \otimes \alpha_2 \otimes \alpha_3$ for $(\alpha_1 \otimes \alpha_2) \otimes \alpha_3$). Similarly, we adopt left-associative notation for pairs of states (e.g., we write (q_1, q_2, q_3) for $((q_1, q_2), q_3)$). Behaviorally, bracketing is insignificant, because $\otimes$ is associative and commutative modulo behavioral congruence as stated in the previous theorems. However, bracketing does matter for our structural reasoning about constraint automata w/MC later in this paper.

Figs. 8 and 9 show an example of multiplication. As shown in this example, to compute the global constraint automaton w/MC for a connector, we compute the product of the local constraint automata w/MC for its nodes and channels using $\otimes$, in an iterative manner. Generally, 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. For instance, in Fig. 8b, let the numbers in the top-left corners indicate values for i in α_i. Then, the computation in Figs. 8 and 9 can be expressed as follows:

$$(\alpha_1 \otimes \alpha_2 \otimes \alpha_3) \otimes \alpha_4 \otimes (\alpha_5 \otimes \alpha_6 \otimes \alpha_7) \otimes \alpha_8 \otimes (\alpha_9 \otimes \alpha_{10} \otimes \alpha_{11}) \otimes \alpha_{12} \otimes \alpha_{13}$$

(Only for the sake of presentation, we do not compute the multiplication in the exact order of the indices, as expressed with the bracketing; because multiplication is associative and commutative, however, this does not matter for the end result.) Henceforth, we call every $\alpha \otimes \alpha_{i<n}$ in this computation an *intermediate composite*; we call $\alpha \otimes \alpha_n$ the *final composite*.

Essentially, multiplication joins the constituent components modeled by its operands on their shared ports. In particular, every execution step involving a shared port by one component must synchronize with every execution step involving the same shared port by the other component. This kind of (synchronous parallel) composition has two notable properties: *multiparty synchronization* and *indirect synchronization*. Multiparty synchronization means that through successive applications, $\otimes$ can synchronize transitions in one constraint automaton w/MC with transitions in multiple other constraint automata w/MC. For instance, $\otimes$ synchronizes the transitions in the constraint automata w/MC in the middle "column" in Fig. 8b with transitions in constraint automata w/MC in both the left column and the right column (i.e., from Fig. 8b to Fig. 9a). Indirect synchronization means that through successive applications, $\otimes$ can synchronize transitions in a constraint automaton w/MC with transitions in another constraint automaton w/MC via intermediate products. For instance, $\otimes$ synchronizes the transitions in the constraint automata w/MC on the top "row" in Fig. 8b with transitions in the constraint automata w/MC on the



(a) Connector (b) Progress: 0/12

Fig. 8. Multiplication example: Alternator$_3$ (part 1/2).

bottom row in Fig. 8b via transitions in the constraint automata w/MC on the middle row in Fig. 8b (i.e., from Fig. 8b to Fig. 9b via Fig. 9a). Indirect synchronization enables compositional construction of globally synchronous composites out of locally synchronous primitives.

The following theorems state that $\simeq$ is a congruence for $\otimes$.

Theorem 7. $[\alpha_1 \simeq \alpha_2$ ***and*** $\alpha_3 \simeq \alpha_4]$ ***implies*** $\alpha_1 \otimes \alpha_3 \simeq \alpha_2 \otimes \alpha_4$

Remark. The main difference between the join operation on constraint automata (without memory cells) and the multiplication on our constraint automata w/MC is the number of rules to derive new transitions with [4]: the join operation on constraint automata (without memory cells) has two additional rules that explicitly state that α_i ($i \in \{0, 1\}$) can fire a transition without shared ports always independent of α_j ($j \in \{0, 1\} \setminus \{i\}$). Effectively, these rules hard-code an assumption that components can always run at their own pace and may idle at will. As already explained in Section 3.3, this assumption is natural in the context of Reo but may generally be too restrictive. In particular, in this paper, we follow Clarke et al. [34], and require idling to be an explicit part of constraint automata w/MC in the form of idling transitions—optionally, if desired.

5.3. Subtraction

Subtraction consumes two constraint automata w/MC α_1 and α_2 as input and produces a constraint automaton w/MC as output. As multiplication, we formally define subtraction on constraint automata w/MC as a partial function. Operationally, to subtract α_2 from α_1, we remove all states, ports, memory cells, transitions, and initial states in α_2 from α_1. Additionally, we also remove all ports in α_2 from all synchronization constraints and data constraints in α_1. Henceforth, let data constraint $\exists\{x_1, \ldots, x_l\}.\phi$ be a shorthand for $\exists x_1. \cdots .\exists x_l.\phi$.

Definition 23 *(subtraction).* $\ominus : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ denotes the partial function defined by the following equation:

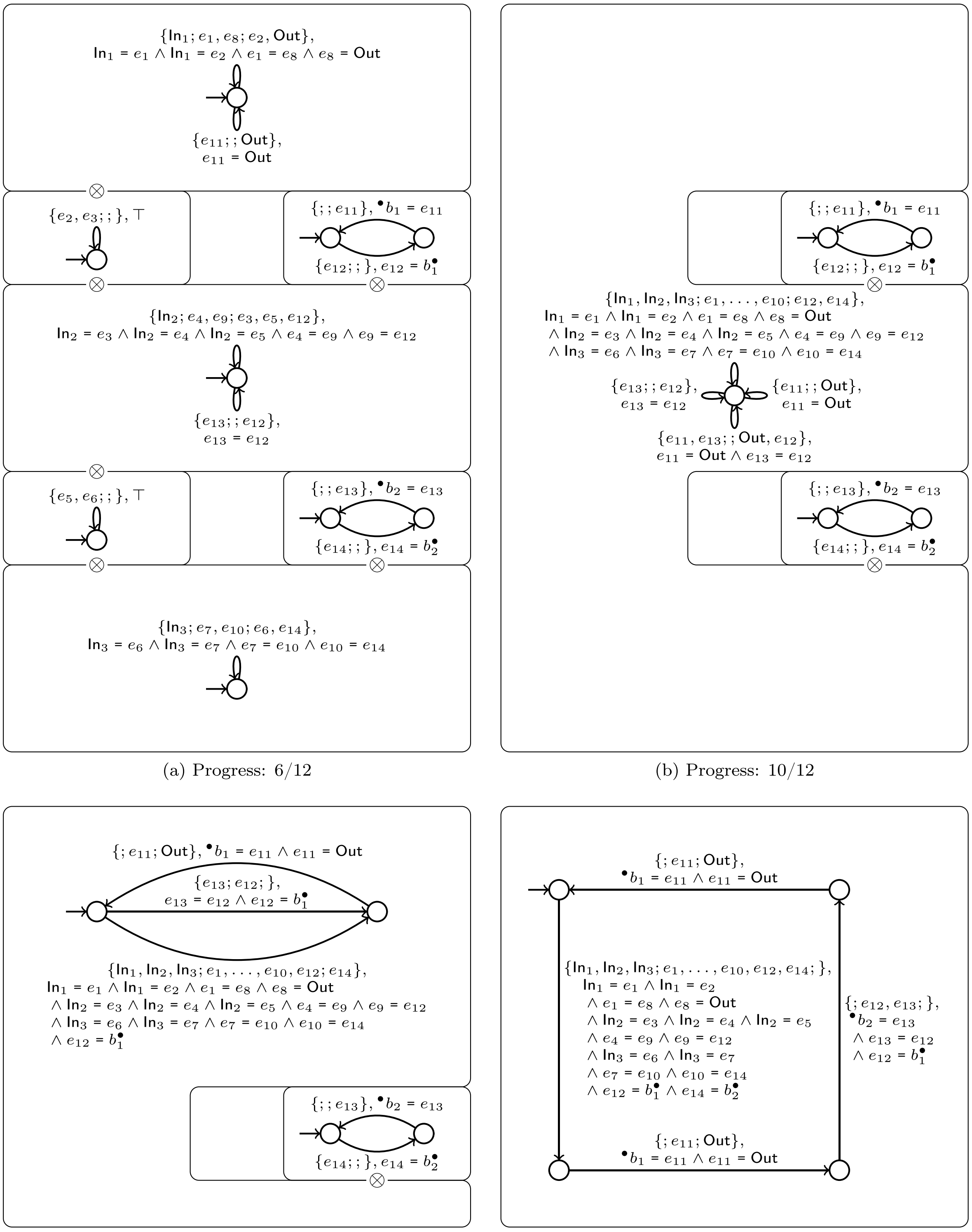

$\{\mathsf{In}_1; e_1, e_8; e_2, \mathsf{Out}\}$,
$\mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2 \wedge e_1 = e_8 \wedge e_8 = \mathsf{Out}$
$\{e_{11};;\mathsf{Out}\}$, $e_{11} = \mathsf{Out}$
$\{e_2, e_3;;\}, \top$
$\{;;e_{11}\}, {}^\bullet b_1 = e_{11}$
$\{e_{12};;\}, e_{12} = b_1^\bullet$
$\{\mathsf{In}_2; e_4, e_9; e_3, e_5, e_{12}\}$,
$\mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5 \wedge e_4 = e_9 \wedge e_9 = e_{12}$
$\{e_{13};;e_{12}\}$, $e_{13} = e_{12}$
$\{e_5, e_6;;\}, \top$
$\{;;e_{13}\}, {}^\bullet b_2 = e_{13}$
$\{e_{14};;\}, e_{14} = b_2^\bullet$
$\{\mathsf{In}_3; e_7, e_{10}; e_6, e_{14}\}$,
$\mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7 \wedge e_7 = e_{10} \wedge e_{10} = e_{14}$
(a) Progress: 6/12
$\{\mathsf{In}_1, \mathsf{In}_2, \mathsf{In}_3; e_1, \ldots, e_{10}; e_{12}, e_{14}\}$,
$\mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2 \wedge e_1 = e_8 \wedge e_8 = \mathsf{Out}$
$\wedge\, \mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5 \wedge e_4 = e_9 \wedge e_9 = e_{12}$
$\wedge\, \mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7 \wedge e_7 = e_{10} \wedge e_{10} = e_{14}$
$\{e_{13};;e_{12}\}$, $e_{13} = e_{12}$
$\{e_{11};;\mathsf{Out}\}$, $e_{11} = \mathsf{Out}$
$\{e_{11}, e_{13};;\mathsf{Out}, e_{12}\}$,
$e_{11} = \mathsf{Out} \wedge e_{13} = e_{12}$
(b) Progress: 10/12
$\{;e_{11};\mathsf{Out}\}, {}^\bullet b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
$\{e_{13};e_{12};\}$,
$e_{13} = e_{12} \wedge e_{12} = b_1^\bullet$
$\{\mathsf{In}_1, \mathsf{In}_2, \mathsf{In}_3; e_1, \ldots, e_{10}, e_{12}; e_{14}\}$,
$\mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2 \wedge e_1 = e_8 \wedge e_8 = \mathsf{Out}$
$\wedge\, \mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5 \wedge e_4 = e_9 \wedge e_9 = e_{12}$
$\wedge\, \mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7 \wedge e_7 = e_{10} \wedge e_{10} = e_{14}$
$\wedge\, e_{12} = b_1^\bullet$
(c) Progress: 11/12
$\{;e_{11};\mathsf{Out}\}$,
${}^\bullet b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
$\{\mathsf{In}_1, \mathsf{In}_2, \mathsf{In}_3; e_1, \ldots, e_{10}, e_{12}, e_{14};\}$,
$\mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2$
$\wedge\, e_1 = e_8 \wedge e_8 = \mathsf{Out}$
$\wedge\, \mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5$
$\wedge\, e_4 = e_9 \wedge e_9 = e_{12}$
$\wedge\, \mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7$
$\wedge\, e_7 = e_{10} \wedge e_{10} = e_{14}$
$\wedge\, e_{12} = b_1^\bullet \wedge e_{14} = b_2^\bullet$
$\{;e_{12}, e_{13};\}$,
${}^\bullet b_2 = e_{13}$
$\wedge\, e_{13} = e_{12}$
$\wedge\, e_{12} = b_1^\bullet$
(d) Progress: 12/12

Fig. 9. Multiplication example: $\mathsf{Alternator}_3$ (part 2/2).

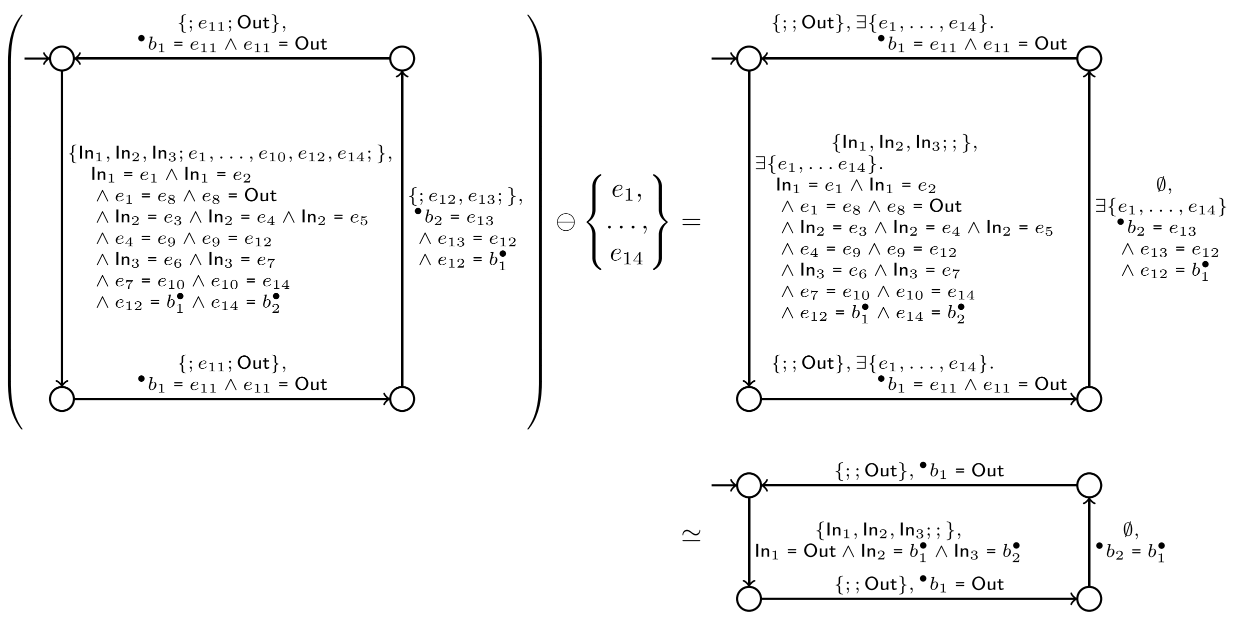

$\{;e_{11};\mathsf{Out}\}$, ${}^{\bullet}b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
$\{\mathsf{In}_1,\mathsf{In}_2,\mathsf{In}_3;e_1,\ldots,e_{10},e_{12},e_{14};\}$, $\mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2 \wedge e_1 = e_8 \wedge e_8 = \mathsf{Out} \wedge \mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5 \wedge e_4 = e_9 \wedge e_9 = e_{12} \wedge \mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7 \wedge e_7 = e_{10} \wedge e_{10} = e_{14} \wedge e_{12} = b_1^{\bullet} \wedge e_{14} = b_2^{\bullet}$
$\{;e_{12},e_{13};\}$, ${}^{\bullet}b_2 = e_{13} \wedge e_{13} = e_{12} \wedge e_{12} = b_1^{\bullet}$
$\{;e_{11};\mathsf{Out}\}$, ${}^{\bullet}b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
$\ominus \left\{ e_1, \ldots, e_{14} \right\} =$
$\{;;\mathsf{Out}\}, \exists\{e_1,\ldots,e_{14}\}.\ {}^{\bullet}b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
$\{\mathsf{In}_1,\mathsf{In}_2,\mathsf{In}_3;;\}$, $\exists\{e_1,\ldots e_{14}\}.\ \mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2 \wedge e_1 = e_8 \wedge e_8 = \mathsf{Out} \wedge \mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5 \wedge e_4 = e_9 \wedge e_9 = e_{12} \wedge \mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7 \wedge e_7 = e_{10} \wedge e_{10} = e_{14} \wedge e_{12} = b_1^{\bullet} \wedge e_{14} = b_2^{\bullet}$
$\emptyset$, $\exists\{e_1,\ldots,e_{14}\}\ {}^{\bullet}b_2 = e_{13} \wedge e_{13} = e_{12} \wedge e_{12} = b_1^{\bullet}$
$\{;;\mathsf{Out}\}, \exists\{e_1,\ldots,e_{14}\}.\ {}^{\bullet}b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
$\simeq$
$\{;;\mathsf{Out}\}, {}^{\bullet}b_1 = \mathsf{Out}$
$\{\mathsf{In}_1,\mathsf{In}_2,\mathsf{In}_3;;\}$, $\mathsf{In}_1 = \mathsf{Out} \wedge \mathsf{In}_2 = b_1^{\bullet} \wedge \mathsf{In}_3 = b_2^{\bullet}$
$\emptyset$, ${}^{\bullet}b_2 = b_1^{\bullet}$
$\{;;\mathsf{Out}\}, {}^{\bullet}b_1 = \mathsf{Out}$

Fig. 10. Subtraction example: Alternator$_3$.

$$\begin{pmatrix} Q_1, \\ \begin{pmatrix} P_1^{\text{all}}, \\ P_1^{\text{in}}, \\ P_1^{\text{out}} \end{pmatrix}, \\ M_1, \\ \longrightarrow_1, \\ Q_1^0 \end{pmatrix} \ominus \begin{pmatrix} Q_2, \\ \begin{pmatrix} P_2^{\text{all}}, \\ P_2^{\text{in}}, \\ P_2^{\text{out}} \end{pmatrix}, \\ M_2, \\ \longrightarrow_2, \\ Q_2^0 \end{pmatrix} = \begin{pmatrix} Q_1 \setminus Q_2, \\ \begin{pmatrix} P_1^{\text{all}} \setminus P_2^{\text{all}}, \\ P_1^{\text{in}} \setminus P_2^{\text{in}}, \\ P_1^{\text{out}} \setminus P_2^{\text{out}} \end{pmatrix}, \\ M_1 \setminus M_2, \\ \longrightarrow_\ominus, \\ Q_1^0 \setminus Q_2^0 \end{pmatrix} \quad \textbf{if} \begin{bmatrix} (P_1^{\text{all}} \setminus P_1^{\text{in}}) \cap P_2^{\text{in}} = \emptyset \\ \textbf{and } (P_1^{\text{all}} \setminus P_1^{\text{out}}) \cap P_2^{\text{out}} = \emptyset \end{bmatrix}$$

where $\longrightarrow_\ominus$ denotes the smallest relation induced by the following rule:

$$\frac{q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \textbf{ and } q_1, q_1' \notin Q_2}{q \xrightarrow{P_1 \setminus P_2^{\text{all}}, \exists P_2^{\text{all}} \cup {}^{\bullet}M_2 \cup M_2^{\bullet}.\phi_1}_\ominus q'} \tag{16}$$

Note that $\simeq$ *is not* a congruence for $\ominus$ in general. As a simple witness, let

$$\begin{aligned} \alpha_1 &= (\{\mathsf{q}_1\}, (\{\mathsf{A}\}, \emptyset, \emptyset), \emptyset, \{(\mathsf{q}_1, \{\mathsf{A}\}, \top, \mathsf{q}_1)\}, \{\mathsf{q}_1\}) \\ \alpha_2 &= (\{\mathsf{q}_2\}, (\{\mathsf{A}\}, \emptyset, \emptyset), \emptyset, \{(\mathsf{q}_2, \{\mathsf{A}\}, \top, \mathsf{q}_2)\}, \{\mathsf{q}_2\}) \\ \alpha\ &= (\{\mathsf{q}_1\}, (\emptyset, \emptyset, \emptyset), \emptyset, \emptyset, \emptyset) \end{aligned}$$

be constraint automata w/mc, where q_1 and q_2 are concrete states from $\mathbb{Q}$. Clearly, $\alpha_1 \simeq \alpha_2$. Subtraction of α from α_1 and α_2, however, yields an empty constraint automaton w/mc and α_2. Clearly, thus, $\alpha_1 \ominus \alpha \not\simeq \alpha_2 \ominus \alpha$. Although $\simeq$ is not a congruence for $\ominus$ in general, it *is* a congruence in all currently practically relevant cases: typically, we do not use the full power of subtraction but only its ability to remove behaviorally insignificant internal ports from synchronization constraints and data constraints, to compress the minuend constraint automaton w/mc. Because this use of subtraction is so important, we overload operation symbol $\ominus$ as follows.

Definition 24 *(subtraction of ports).* $\ominus : \mathbb{A}\text{UTOM} \times 2^{\mathbb{P}} \rightarrow \mathbb{A}\text{UTOM}$ denotes the function defined by the following equation:

$$\alpha \ominus P = \alpha \ominus (\emptyset, (P, P \cap \mathsf{Input}(\alpha), P \cap \mathsf{Outp}(\alpha)), \emptyset, \emptyset, \emptyset)$$

Note that subtraction of constraint automata w/mc is partial whereas subtraction of ports is total.

Fig. 10 shows an example of subtraction of (internal) ports. The constraint automaton w/mc on the left-hand side of the equality symbol in Fig. 10 is the same as the final composite in Fig. 9d. The constraint automaton w/mc on the right-hand side of the equality symbol in Fig. 10 and the constraint automaton w/mc on the second line in Fig. 10 are congruent, because substitution of data constraints with equivalent data constraints—the only difference between these two constraint automata w/mc—preserves congruence (as one would expect). Note also that the constraint automaton w/mc on the second line in Fig. 10 and the constraint automaton W/MC in Fig. 6 already look quite alike; to make them equal, we need just one more operation to also hide the internal transition in the former constraint automaton W/MC (from its third state to its fourth state).

The following theorems state that $\simeq$ is a congruence for $\ominus$ and that subtraction of unshared ports distributes over multiplication.

Theorem 8. $\alpha_1 \simeq \alpha_2$ ***implies*** $\alpha_1 \ominus P \simeq \alpha_2 \ominus P$

Theorem 9. $P \cap \mathsf{Port}(\alpha_1) \cap \mathsf{Port}(\alpha_2) = \emptyset$ ***implies*** $(\alpha_1 \otimes \alpha_2) \ominus P \simeq (\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)$

Remark. The main difference between the hide operation on constraint automata (without memory cells) and the subtraction of ports on our constraint automata W/MC is that the hide operation on constraint automata (without memory cells) also aggregates internal transitions. Because aggregation is substantially more complex in constraint automata W/MC, we present aggregation as a separate operation in Section 5.4.

5.4. Aggregation

The idea behind aggregation is simple: replace every chain of internal transitions followed by an observable transition, $q_1 \xrightarrow{\emptyset,\phi_1} q_2 \xrightarrow{\emptyset,\phi_2} \cdots \xrightarrow{\emptyset,\phi_{n-1}} q_n \xrightarrow{P,\phi_n} q_{n+1}$ where $P \neq \emptyset$, with a transition from q_1 to q_{n+1} labeled with synchronization constraint P and *some combination* of data constraints $\phi_1, \ldots \phi_{n+1}$. This new transition, then, all by itself models a sequence of internal execution steps of a component followed by an observable one. In the absence of memory cells, data constraints can straightforwardly be combined by taking their conjunction, as data constraints without memory cells have no "sequential dependencies" among each other. Data constraints with memory cells do have such dependencies. Consider, for instance, transitions $\circ \xrightarrow{\emptyset,{}^\bullet b_2 = b_1^\bullet} \circ \xrightarrow{\{;;\mathrm{Out}\},{}^\bullet b_1 = \mathrm{Out}} \circ$ in the constraint automaton W/MC on the second line of Fig. 10. The sequential dependency between the data constraints of these two transitions is that $b_1^\bullet$ in the first data constraint and ${}^\bullet b_1$ in the second data constraint actually refer to the same datum: in the first transition, a datum flows into buffer b_1, after which in the second transition, that same datum flows out of b_1. Replacing these two transitions with transition $\circ \xrightarrow{\{;;\mathrm{Out}\},{}^\bullet b_2 = b_1^\bullet \wedge {}^\bullet b_1 = \mathrm{Out}} \circ$ is thus incorrect. Instead, before taking the conjunction, we should manipulate data constraints so as to preserve their sequential dependencies. For instance, let $\dagger$ denote some fresh data variable (relative to the two data constraints under consideration); a correct replacement transition in our example, then, is $\circ \xrightarrow{\{;;\mathrm{Out}\},\exists\dagger.({}^\bullet b_2 = \dagger \wedge \dagger = \mathrm{Out})} \circ$, which can be further simplified to $\circ \xrightarrow{\{;;\mathrm{Out}\},{}^\bullet b_2 = \mathrm{Out}} \circ$.

To generalize the previous example, we need to introduce some technical machinery. First, we more generally define $\dagger$ as an injection that consumes as input a memory cell and a natural number and produces as output a data variable. We stipulate that this output data variable is fresh relative to the data constraints under consideration (it is possible to formalize this freshness, but we skip doing so here for simplicity).

Definition 25 *(memory cell substitutes).* $\dagger : \mathbb{M} \times \mathbb{N} \to \mathbb{X} \setminus ({}^\bullet\mathbb{M} \cup \mathbb{M}^\bullet)$ is an injective function.

Next, we introduce three functions that substitute $\dagger_i(m)$ for ${}^\bullet m$ and $m^\bullet$ data constraints. Let $\phi\Sigma$ denote the application of every substitution $[x'/x]$ in Σ to ϕ (assuming that every x' differs from every x to avoid ambiguity, which is covered by our previous freshness assumption).

Definition 26 *(substitution of memory cells).* $\mathsf{substNext}, \mathsf{substPrev}, \mathsf{subst} : \mathbb{DC} \to \mathbb{DC}$ denote the functions defined by the following equations:

$$\begin{aligned}
\mathsf{substNext}_i(\phi) &= \phi\{[\dagger_i(m)/m^\bullet] \mid m \in \mathbb{M}\} \\
\mathsf{substPrev}_i(\phi) &= \phi\{[\dagger_{i-1}(m)/{}^\bullet m] \mid m \in \mathbb{M}\} \\
\mathsf{subst}_i(\phi) &= \mathsf{substNext}_i(\mathsf{substPrev}_i(\phi))
\end{aligned}$$

Now, aggregation consumes as input a constraint automaton W/MC and produces as output a constraint automaton W/MC. We formally define aggregation on constraint automata W/MC as a total function.

Definition 27 *(aggregation).* $\cdot^\star : \mathbb{A}\mathrm{UTOM} \to \mathbb{A}\mathrm{UTOM}$ denotes the function defined by the following equation:

$$(Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow, Q^0)^\star = (Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow_\star, Q^0)$$

where $\longrightarrow_\star$ denotes the smallest relation defined by the following rules:

$$\frac{q_1 \xrightarrow{\emptyset,\phi_1} q_2 \xrightarrow{\emptyset,\phi_2} \cdots \xrightarrow{\emptyset,\phi_{n-1}} q_n \xrightarrow{P,\phi_n} q_{n+1} \textbf{ and } P \neq \emptyset}{q_1 \xrightarrow{P,\exists \mathsf{Img}(\dagger).\mathsf{substNext}_1(\phi_1) \wedge \mathsf{subst}_2(\phi_2) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{n-1}) \wedge \mathsf{substPrev}_n(\phi_n)}_\star q_{n+1}} \qquad \frac{q \xrightarrow{P,\phi} q' \textbf{ and } P \neq \emptyset}{q \xrightarrow{P,\phi}_\star q'} \tag{17}$$

In this definition, $q_1 \xrightarrow{\emptyset,\phi_1} q_2 \xrightarrow{\emptyset,\phi_2} \cdots \xrightarrow{\emptyset,\phi_{n-1}} q_n \xrightarrow{P,\phi_n} q_{n+1}$ means that there exists a natural number $n \geq 1$ such that (i) $q_1 \xrightarrow{\emptyset,\phi_1} q_2$, (ii) for all $1 \leq i < n$, there exist a state q_i and a data constraint ϕ_i such that $q_i \xrightarrow{\emptyset,\phi_i} q_{i+1}$, and (iii) $q_n \xrightarrow{P,\phi_{n+1}} q_{n+1}$. The meaning of the ellipsis in the consequent of the rule is that the conjunction contains a conjunct $\mathsf{subst}_i(\phi_i)$ for every ϕ_i in the antecedent of the rule.

If we apply aggregation to the constraint automaton w/MC on the second line in Fig. 10, and if we simplify the resulting constraint automaton w/MC modulo data constraint equivalence, we obtain exactly the same constraint automaton w/MC as shown in Fig. 6. This concludes our compositional construction of the constraint automaton w/MC for $\mathsf{Alternator}_3$.

The following theorem states that $\simeq$ is a congruence for $\cdot^\star$.

Theorem 10. $\alpha_1 \simeq \alpha_2$ ***implies*** $\alpha_1^\star \simeq \alpha_2^\star$

In addition to the previous congruence theorem, the following two theorems are essential to safely use aggregation in practice. The first theorem states that aggregation does not change the behavior of a constraint automaton w/MC. The second theorem states that aggregation is a homomorphism for multiplication.

Theorem 11. $\alpha \approx \alpha^\star$

Theorem 12. $[\mathsf{IdlingEnabled}(\alpha_1)$ ***and*** $\mathsf{IdlingEnabled}(\alpha_2)]$ ***implies*** $(\alpha_1 \otimes \alpha_2)^\star \simeq \alpha_1^\star \otimes \alpha_2^\star$

Note that in Theorem 11, we use behavioral equivalence instead of behavioral congruence. Behavioral congruence is too strong in this case, because it is sensitive to internal transitions, while aggregation exactly removes such transitions. Behavioral equivalence, in contrast, is insensitive to internal transitions.

Remark. As already remarked in Section 5.3, the hide operation on constraint automata (without memory cells) combines both subtraction of ports and aggregation. However, aggregation in constraint automata (without memory cells) is significantly simpler due to the absence of memory cells (i.e., data constraints in a chain of transitions in a constraint automaton (without memory cells) can be straightforwardly combined using conjunction, without first applying substitutions).

5.5. *Pruning*

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. When using the operations on constraint automata w/MC defined in this section, we can obtain a constraint automaton w/MC that contains states that are unreachable, even if the constraint automata w/MC we started out with do not have this property. This can occur for one of three reasons:

1. *By multiplication*: a state $(q_1, q_2) \in \mathsf{Stat}(\alpha_1 \otimes \alpha_2)$ is unreachable if there is no pair of equally long paths of length k from $\mathsf{Init}(\alpha_1)$ to q_1 in α_1 and $\mathsf{Init}(\alpha_2)$ to q_2 in α_2 such that, for all $1 \leq i \leq k$, the i-th transition on the path in α_1 and the i-th transition on the path in α_2 can compose into a new transition (which requires their synchronization and the data constraints to agree), i.e., such that they manifest as a path from $\mathsf{Init}(\alpha_1) \times \mathsf{Init}(\alpha_2)$ to (q_1, q_2) in $\alpha_1 \otimes \alpha_2$.
2. *By aggregation*: when all incoming transitions to $q \in \mathsf{Stat}(\alpha)$ are internal, q is unreachable in $\alpha^\star$.
3. *By transitivity*: a state q may be unreachable because it is only reachable from states that have become unreachable as a result of one of the two cases above; when this is the case, we attribute the unreachability of q to its root cause.

We formalize *pruning* as an operation on constraint automata w/MC that removes unreachable states. Let Reach denote the *reachability function* that consumes as input a constraint automaton w/MC and produces as output its set of reachable states.

Definition 28. $\mathsf{Reach} : \mathbb{A}\textsc{utom} \to 2^{\mathbb{Q}}$ is the function defined as follows[4]:

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

[4] Alternatively, one can define Reach inductively as follows:

$$\begin{array}{ll} \mathsf{Reach}(\alpha, q) & = \bigcup\{\{q'\} \cup \mathsf{Reach}(\alpha, q') \mid (q, P, \phi, q') \in \mathsf{Trans}(\alpha)\} \\ \mathsf{Reach}(\alpha) & = \bigcup\{\{q_0\} \cup \mathsf{Reach}(\alpha, q_0) \mid q_0 \in \mathsf{Init}(\alpha)\}. \end{array}$$

In this definition, $(q_1, P_1, \phi_1, q_2), \ldots, (q_{k-1}, P_{k-1}, \phi_{k-1}, q_k) \in \mathsf{Trans}(\alpha)$ means that for all $1 \le i < k$, there exist states q_i, q_{i+1}, a set of ports P_i and a data constraint ϕ_i such that $(q_i, P_i, \phi_i, q_i) \in \mathsf{Trans}(\alpha)$.

Next, let $\lfloor\cdot\rfloor$ denote the *pruning function*, which takes as input a constraint automaton w/MC and produces as output a behaviorally congruent—proven below—constraint automaton w/MC for its reachable states (i.e., the pruning function prunes a constraint automaton w/MC down to its reachable fragment).

Definition 29. $\lfloor\cdot\rfloor : \mathbb{A}\text{UTOM} \to \mathbb{A}\text{UTOM}$ is the function defined as follows:

$$\lfloor\alpha\rfloor = (\mathsf{Reach}(\alpha), (\mathsf{Port}(\alpha), \mathsf{Input}(\alpha), \mathsf{Outp}(\alpha)), \mathsf{Memor}(\alpha), \longrightarrow_{\lfloor\cdot\rfloor}, \mathsf{Init}(\alpha))$$

where $\longrightarrow_{\lfloor\cdot\rfloor}$ denotes the smallest relation induced by the rule

$$\frac{(q, P, \phi, q') \in \mathsf{Trans}(\alpha) \textbf{ and } q \in \mathsf{Reach}(\alpha)}{q \xrightarrow{P,\phi}_{\lfloor\cdot\rfloor} q'} \tag{18}$$

The following theorem states that a constraint automaton w/MC and its pruned version are behaviorally congruent.

Theorem 13. $\alpha \simeq_{\{(q,q) \mid q \in \mathsf{Reach}(\alpha)\}} \lfloor\alpha\rfloor$

6. Composition approach I: automaton-by-automaton

6.1. Approach

Suppose that we have n constraint automata w/MC that we must compose into one (e.g., our Reo-to-Java compiler needs to compute the global constraint automaton w/MC for a connector from the local constraint automata w/MC for that connector's nodes and channels). Arguably the most natural approach to do this—our Reo-to-Java compiler has been using this approach from its inception—is the *automaton-by-automaton* approach: to compose constraint automata w/MC $\alpha_1, \ldots, \alpha_n$ using this approach, we first compose α_1 with α_2, then we compose the resulting (intermediate) composite with α_3, and so on. Every composition step in this process consists of multiplication, subtraction of ports, aggregation, and pruning, as follows:

$$\underbrace{\lfloor((\cdots(\underbrace{\lfloor((\underbrace{\lfloor((\alpha_1 \otimes \alpha_2) \ominus P_{12})^\star\rfloor}_{\text{composition 1}} \otimes \alpha_3) \ominus P_{(12)3})^\star\rfloor}_{\text{composition 2}} \cdots) \otimes \alpha_n) \ominus P_{((12)3)\cdots n})^\star\rfloor}_{\text{composition } n-1}$$

where P_{12} contains the internal ports of $\alpha_1 \otimes \alpha_2$, $P_{(12)3}$ the internal ports of $((\alpha_1 \otimes \alpha_2) \ominus P_{12}) \otimes \alpha_3$, etc.

Though conceptually simple, there is a subtle—and easy to miss—point about use of aggregation in the automaton-by-automaton approach to composition. Recall that Theorem 11 states that aggregation preserves behavioral equivalence but *not* behavioral congruence. As we are using aggregation inside multiplication, Theorem 11 is thus insufficient: generally, we cannot conclude $\alpha_1 \otimes \alpha_2 \approx \alpha_1^\star \otimes \alpha_2$ from $\alpha_1 \approx \alpha_1^\star$. To resolve this issue, we define *another* congruence beside behavioral congruence, called *weak behavioral congruence*, that implies behavioral equivalence (as behavioral congruence does), that is a congruence for multiplication (as behavioral congruence is), and that equates α to $\alpha^\star$ (as behavioral equivalence does).

Definition 30 *(weak behavioral congruence).* $\approxeq \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ is the smallest relation induced by the following rule:

$$\frac{\alpha_1^\star \simeq \alpha_2^\star}{\alpha_1 \approxeq \alpha_2} \tag{19}$$

Theorem 14. $\approxeq$ *is an equivalence*

The following two theorems state that behavioral congruence implies weak behavioral congruence (to motivate the "weak") and that weak behavioral congruence implies behavioral equivalence (cf. Theorem 4).

Theorem 15. $\simeq \subseteq \approxeq$

Theorem 16. $\approxeq \subseteq \approx$

The following two theorems state that $\approxeq$ is a congruence for multiplication (cf. Theorem 7) and that aggregation preserves weak behavioral congruence (cf. Theorem 11; the correctness criterion for aggregation).

Theorem 17. $\left[\begin{array}{l}\alpha_1 \approxeq \alpha_2 \textbf{ and } \alpha_3 \approxeq \alpha_4 \\ \textbf{and } \mathsf{IdlingEnabled}(\alpha_1 \otimes \alpha_3) \\ \textbf{and } \mathsf{IdlingEnabled}(\alpha_2 \otimes \alpha_4)\end{array}\right]$ **implies** $\alpha_1 \otimes \alpha_3 \approxeq \alpha_2 \otimes \alpha_4$

Theorem 18. $\alpha \approxeq \alpha^\star$

Condition $\mathsf{IdlingEnabled}(\alpha_1 \otimes \alpha_3)$ in Theorem 17 is equivalent to $\left[\mathsf{IdlingEnabled}(\alpha_1) \textbf{ and } \mathsf{IdlingEnabled}(\alpha_3)\right]$; multiplication preserves idling-enabledness. Note also that Theorem 17 works only for idling-enabled constraint automata w/MC. As a simple witness of $\approxeq$ not being a congruence for multiplication in general, let

$$\begin{aligned}
\alpha_1 \quad &= (\{q_1\}, (\{A\}, \emptyset, \emptyset), \emptyset, \{(q_1, \{A\}, \top, q_1)\}, \{q_1\}) \\
\alpha_2 \quad &= (\{q_2, q_2'\}, (\{A\}, \emptyset, \emptyset), \emptyset, \{(q_2, \emptyset, \top, q_2'), (q_2', \{A\}, \top, q_2')\}, \{q_2\}) \\
\alpha_3 = \alpha_4 &= (\{q_3\}, (\{B\}, \emptyset, \emptyset), \emptyset, \{(q_3, \{B\}, \top, q_3)\}, \{q_3\})
\end{aligned}$$

be constraint automata w/MC, where q_1, q_2, q_2' and q_3 are concrete states from $\mathbb{Q}$. Clearly, $\alpha_1 \approxeq \alpha_2$ and $\alpha_3 \approxeq \alpha_4$. However, whereas $\alpha_1 \otimes \alpha_3$ always starts with a $\{A, B\}$-transition, $\alpha_2 \otimes \alpha_4$ always starts with a $\{B\}$-transition.

The final theorem in this section states that first computing all multiplications, then computing all subtractions of ports, and finally computing one aggregation produces a behaviorally equivalent constraint automaton w/MC to the constraint automaton w/MC computed in the automaton-by-automaton approach (where multiplying, subtracting, aggregating, and pruning are interleaved).

Theorem 19. *The following weak behavioral congruence holds.*

$$\lfloor((\cdots(\lfloor((\alpha_1 \otimes \alpha_2) \ominus P_{12})^\star\rfloor \cdots) \otimes \alpha_n) \ominus P_{(12)\cdots n})^\star\rfloor \approxeq \lfloor((\alpha_1 \otimes \cdots \otimes \alpha_n) \ominus P_{12} \ominus \cdots \ominus P_{(12)\cdots n})^\star\rfloor$$

In this theorem, we use the following shorthands:

- $\lfloor((\cdots(\lfloor((\alpha_1 \otimes \alpha_2) \ominus P_{12})^\star\rfloor \cdots) \otimes \alpha_n) \ominus P_{(12)\cdots n})^\star\rfloor$ is a shorthand for $f(\alpha_1, \ldots, \alpha_n)$, where $\alpha_1, \ldots, \alpha_n$ is a shorthand for a sequence of n alphas, and where $f : \bigcup\{\mathbb{A}\textsc{utom}^i \to \mathbb{A}\textsc{utom} \mid i > 0\}$ is recursively defined by the following equation:

$$f(\alpha_1, \ldots, \alpha_n) = \begin{cases} \alpha_1 & \textbf{if } n = 1 \\ \lfloor((f(\alpha_1, \ldots, \alpha_{n-1}) \otimes \alpha_n) \ominus (\mathsf{Port}(f(\alpha_1, \ldots, \alpha_{n-1})) \cap \mathsf{Port}(\alpha_n)))^\star\rfloor & \textbf{if } n > 1 \end{cases}$$

- $\alpha_1 \otimes \cdots \otimes \alpha_n$ is a shorthand for $f_\otimes(\alpha_1, \ldots, \alpha_n)$, where $\alpha_1, \ldots, \alpha_n$ is a shorthand for a sequence of n alphas, and where $f_\otimes : \bigcup\{\mathbb{A}\textsc{utom}^i \to \mathbb{A}\textsc{utom} \mid i > 0\}$ is recursively defined by the following equation:

$$f_\otimes(\alpha_1, \ldots, \alpha_n) = \begin{cases} \alpha_1 & \textbf{if } n = 1 \\ f(\alpha_1, \ldots, \alpha_{n-1}) \otimes \alpha_n & \textbf{if } n > 1 \end{cases}$$

- $\alpha \ominus P_1 \ominus \cdots \ominus P_n$ is a shorthand for $f_\ominus(\alpha, P_1, \ldots, P_n)$, where $P_1, \ldots, P_n$ is a shorthand for a sequence of n alphas, and where $f_\ominus : \bigcup\{\mathbb{A}\textsc{utom} \times \mathbb{P}^i \to \mathbb{A}\textsc{utom} \mid i > 0\}$ is recursively defined by the following equation:

$$f_\ominus(\alpha, P_1, \ldots, P_n) = \begin{cases} \alpha \ominus P_1 & \textbf{if } n = 1 \\ f(\alpha, P_1, \ldots, P_{n-1}) \ominus P_n & \textbf{if } n > 1 \end{cases}$$

In the next section, we use the previous theorem to relate the automaton-by-automaton approach to the alternative approach presented.

6.2. Experimental results and analysis

To study the performance of the automaton-by-automaton approach to composition, we performed a number of experiments. In every experiment, we let our Reo-to-Java compiler compute the global constraint automaton w/MC for a connector from the local constraint automata w/MC for that connector's nodes and channels (without subsequently actually generating code) and measured the composition time. To improve performance, our compiler first heuristically orders the local constraint automata w/MC in such a way that, later in the computation, every intermediate composite is always further composed with one of its direct neighbors (if it has any). Generally, this is more efficient than composing constraint automata w/MC randomly.

To perform our experiments, we selected a number of k-parametric *families* of connectors. Every *member* in a family of connectors embodies the same concurrency protocol as the other members in that family; what differs among members—and this is what parameter k represents—is the number of components that can be coordinated by the connector. As such, k controls the size of a connector. Fig. 1 shows the $k = 2$ members of the families with which we experimented. One can extend these $k = 2$ members to $k > 2$ members in a similar way as how we extended Fig. 1a to Fig. 1b (i.e., by duplicating a subconnector and juxtaposing that duplicate with the original connector). We selected these particular families because their members exhibit different behavior in terms of synchrony, exclusion, nondeterminism, direction, sequentiality, and parallelism, thereby aiming for a balanced set of experiments.

Fig. 11 shows the composition times measured for the k-parametric families under study, for $2 \leq k \leq 64$, averaged over sixteen runs.[5] For two families, $\mathsf{EarlyAsyncReplicator}_k$ and $\mathsf{LateAsyncMerger}_k$, the compiler scales roughly linearly in k. This is expected, because the number of transitions of the final composites computed for $\mathsf{EarlyAsyncReplicator}_k$ and $\mathsf{LateAsyncMerger}_k$ also grows linearly in k (while the number of states stays constant). For one family, $\mathsf{Alternator}_k$, the compiler scales roughly quadratically in k, because its final composites, before aggregation, grow quadratically in k. For the remaining six families, the compiler exhausted its available resources (five minutes of time or 2 GB of heap space) long before reaching $k = 64$. This was caused by "rapid"—at least exponential—growth in k. For four of these families, we have a good explanation for why this happened: the final composites computed for $\mathsf{EarlyAsyncMerger}_k$, $\mathsf{EarlyAsyncBarrierMerger}_k$, $\mathsf{LateAsyncReplicator}_k$, and $\mathsf{LateAsyncRouter}_k$ grow exponentially in k, so that the amount of resources required to compute those final composites logically also grows at least exponentially in k. For the other two families, in contrast, our measurements seem more difficult to explain: the final composites computed for $\mathsf{EarlyAsyncOutSequencer}_k$ and Lock_k grow only linearly in k, making an exponential growth in resource requirements rather surprising. The fact that the plots for these two families look identical (the scales on the y-axes are different, but only the shapes/trends of the plots matter here; not the absolute numbers) is a first indication that these families may be suffering from the same issue.

Analysis of the intermediate composites of $\mathsf{EarlyAsyncOutSequencer}_k$ and Lock_k revealed the following: even if final composites grow linearly in k, their intermediate composites, as explicitly computed in the automaton-by-automaton approach, may nevertheless grow exponentially in k. We can explain this easier for $\mathsf{EarlyAsyncOutSequencer}_k$, through the size of its state space, but the same argument applies to Lock_k. $\mathsf{EarlyAsyncOutSequencer}_k$ 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 $\blacksquare$ (its actual value does not matter); the other buffered channels initially contain nothing. As in the literature [1,2], we call this subconnector $\mathsf{Sequencer}_k$. Because no new data can flow into $\mathsf{Sequencer}_k$, only $\blacksquare$ indefinitely cycles through the buffers so that only one buffer holds a datum at any time. Consequently, the constraint automaton W/MC for $\mathsf{Sequencer}_k$ has only k states, each of which represents the presence of $\blacksquare$ in exactly one of its k buffers.

However, if we compose the local constraint automata W/MC for $\mathsf{Sequencer}_k$'s nodes and channels using the automaton-by-automaton 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 constraint automaton W/MC for the largest chain has 2^k states. Only when we compose this penultimate composite with the last local constraint automaton W/MC, the reachable state space collapses into k states, as we "find out" that the open-ended chain actually forms a cycle with exactly one datum; indeed, this is an instance of unreachability-by-multiplication, as explained in Section 5.5. Because $\mathsf{Sequencer}_k$ constitutes $\mathsf{EarlyAsyncOutSequencer}_k$, also $\mathsf{EarlyAsyncOutSequencer}_k$ suffers from this problem.

Fig. 12 shows our previous analysis in pictures. Notably, the intermediate composites in Fig. 12—the first three constraint automata W/MC from the left—contain progressively more states with the following peculiar property: they are reachable from an initial state in those intermediate composites, called *intermediate-reachability*, but neither those states themselves nor any composite state that they constitute, are reachable in the final composite, called *eventual-unreachability*. Thus, by using the automaton-by-automaton approach for computing a final composite, we may spend exponentially many resources on generating a state space that can be pruned without altering its behavior; the computations to find out the structure of this space are, thus, useless. This seems the heart of the problem with the automaton-by-automaton approach and explains the previously surprising performance results for $\mathsf{EarlyAsyncOutSequencer}_k$ and Lock_k.

7. Composition approach II: state-by-state

7.1. Approach

To solve the problem with the automaton-by-automaton approach to composition discovered in Section 6.2, in this section, we present a novel alternative: the *state-by-state* approach. In this alternative approach, instead of composing constraint automata W/MC in sequence, we compose constraint automata W/MC in parallel. It works as follows. We start by computing the initial state of the final composite (by straightforwardly forming the Cartesian product of the sets of initial states in the constraint automata W/MC under composition). Subsequently, we *expand* each of those initial states by computing their outgoing composite transitions. These composite transitions enter new composite states, which we subsequently recursively expand. As such, we compute only the reachable states of the final composite, avoiding the unnecessary computation of now-reachable-but-eventually-unreachable states. Of course, we do not expand composite states that have been expanded before; this may remind one of memoization in dynamic programming.

To formalize the state-by-state approach and its central concepts, we start by formalizing the state-based *decomposition* of a constraint automata W/MC into its per-state "subautomata" and the *recomposition* of that constraint automaton W/MC

[5] Different scales on the y-axes of the plots in Fig. 11 convey an intended message: we mean to compare compile times of individual cases only within each family, but not *across* families. Meaningful as they may be, cross-family comparisons of compile time figures convey information irrelevant for our purpose in this paper.

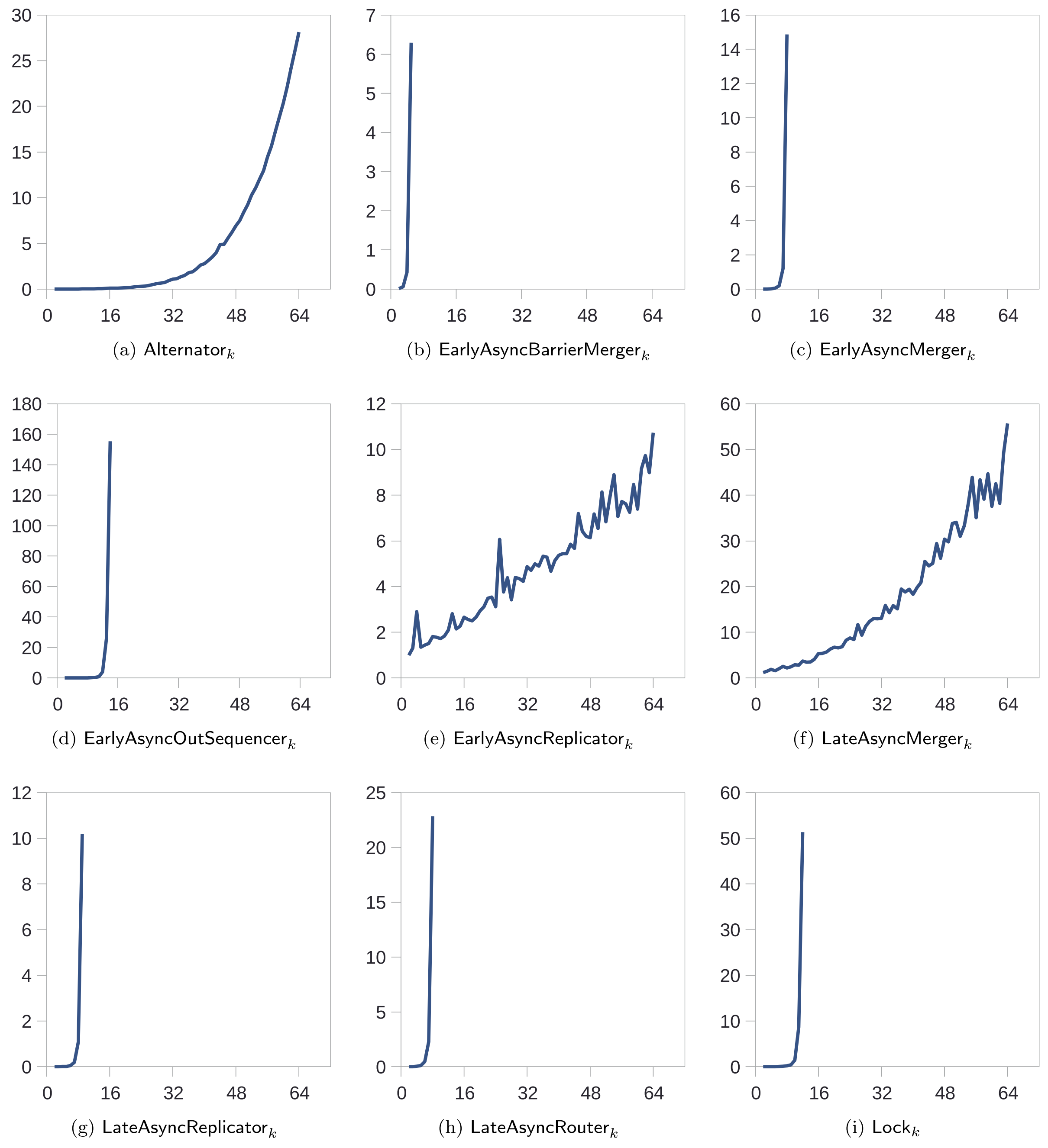


Fig. 11. Composition times (y-axis) for nine k-parametric families, for $2 \le k \le 64$ (x-axis). Time is measured in seconds, except for EarlyAsyncReplicator$_k$ and LateAsyncMerger$_k$, where time is measured in milliseconds.

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 31. $\sigma : 2^{\mathbb{Q}\times 2^{\mathbb{P}}\times\mathbb{DC}\times\mathbb{Q}} \times \mathbb{Q} \rightarrow 2^{\mathbb{Q}\times 2^{\mathbb{P}}\times\mathbb{DC}\times\mathbb{Q}}$ is the function defined as follows:

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

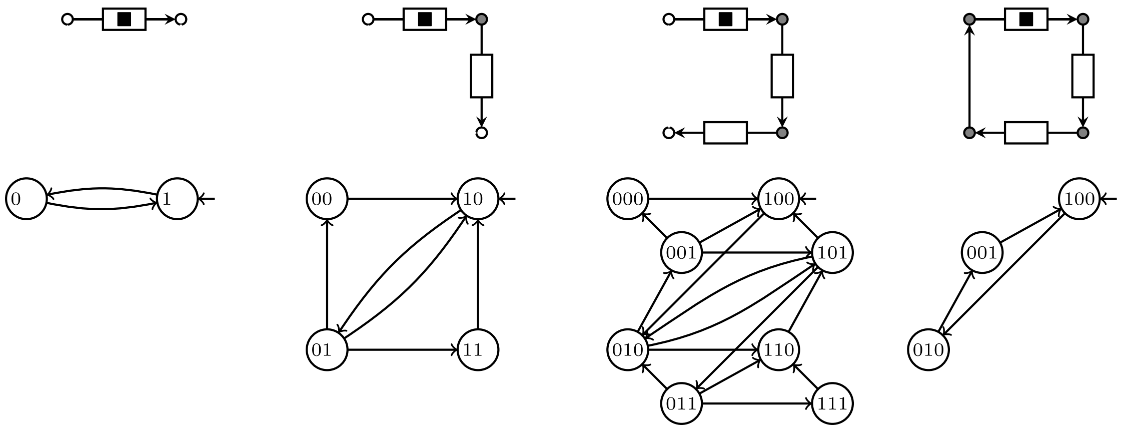


Fig. 12. Composition of the constraint automata w/mc for a cycle of three buffered channels (of capacity 1), closed by a synchronous channel, using the automaton-by-automaton approach. State labels xyz indicate the emptiness/fullness of buffers, where x refers to the first buffer, y to the second buffer, and z to the third buffer; we omitted transition labels to avoid clutter. One buffer is full in the initial state, because otherwise, there is no behavior once the cycle is closed.

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

Definition 32. $\cdot\langle\cdot\rangle : \mathbb{A}\text{UTOM} \times \mathbb{Q} \to \mathbb{A}\text{UTOM}$ is the function defined as follows:

$$\alpha\langle q\rangle = (\mathsf{Stat}(\alpha), (\mathsf{Port}(\alpha), \mathsf{Input}(\alpha), \mathsf{Outp}(\alpha)), \mathsf{Memor}(\alpha), \sigma_q(\mathsf{Trans}(\alpha)), \mathsf{Init}(\alpha))$$

We call q the *significant state* in $\alpha\langle q\rangle$.

To exemplify state-based decomposition—and shortly also recomposition—we revisit our Alternator$_3$ example that we used in Sections 3 and 5 to illustrate (operations on) constraint automata w/mc (see Figs. 6, 8, 9, 10; see also Section 2.2 for an informal description of the behavior of Alternator$_3$). Fig. 13 shows the state-based decomposition of the constraint automata w/mc indexed 12 and 13 in Fig. 8b (i.e., the constraint automata w/mc for the two asynchronous channels that constitute Alternator$_3$). Because the other constraint automata in Fig. 8b have only a single state, state-based decomposition is basically an identity function for them. Fig. 13 shows also the state-based decomposition of the constraint automata w/mc in Fig. 9d.

The following theorem states that decomposition distributes over composition: instead of first computing the composition of n local constraint automata w/mc and then decomposing the resulting global constraint automaton w/mc relative to a global state, we can equally first decompose every local constraint automaton w/mc relative to its local state and then compute the composition of the resulting per-state decompositions.

Theorem 20 *([35], Lemma 13, p. 31).* $(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle(q_1, \ldots, q_n)\rangle = \alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle$

In this theorem, we use the same shorthands as those defined below Theorem 19. Additionally, let $q_1, \ldots, q_n$ denote a sequence of n states.

We now revisit our previous Alternator$_3$ example. The global constraint automaton w/mc for Alternator$_3$ has four global states, as shown in Fig. 9d. The decompositions of that automaton relative to those global states are shown in Figs. 13e–13h. By the previous theorem, we can equally decompose the local constraint automata w/mc that constitute Alternator$_3$ and compose the resulting decompositions. Because all automata in Fig. 8b, except the ones indexed by 12 and 13, have only one state, the composition of those single-state automata with the automata in Figs. 13a (top buffer empty) and 13c (bottom buffer empty) equals the automaton in Fig. 13e (both buffers empty). Likewise, the composition of the single-state automata and the automata in Figs. 13b (top buffer full) and 13d (top buffer full) equals the automaton in Fig. 13f (both buffers full). Likewise, the composition of the single-state automata and the automata in Figs. 13a (top buffer empty) and 13d (bottom buffer full) equals the automaton in Fig. 13g (top buffer empty, bottom buffer full). Likewise, the composition of the single-state automata and the automata in Figs. 13b (top buffer full) and 13d (bottom buffer empty) equals the automaton in Fig. 13h (top buffer full, bottom buffer empty).

The previous definitions (and theorem) cover the essentials of state-based decomposition; next, we discuss recomposition. Let $\bigsqcup$ denote a *recomposition function* that consumes as input a set of constraint automata w/mc and produces as output a constraint automaton w/mc by taking the union of the sets of states, sets of ports, sets of memory cells, sets of transitions, and sets of initial states.

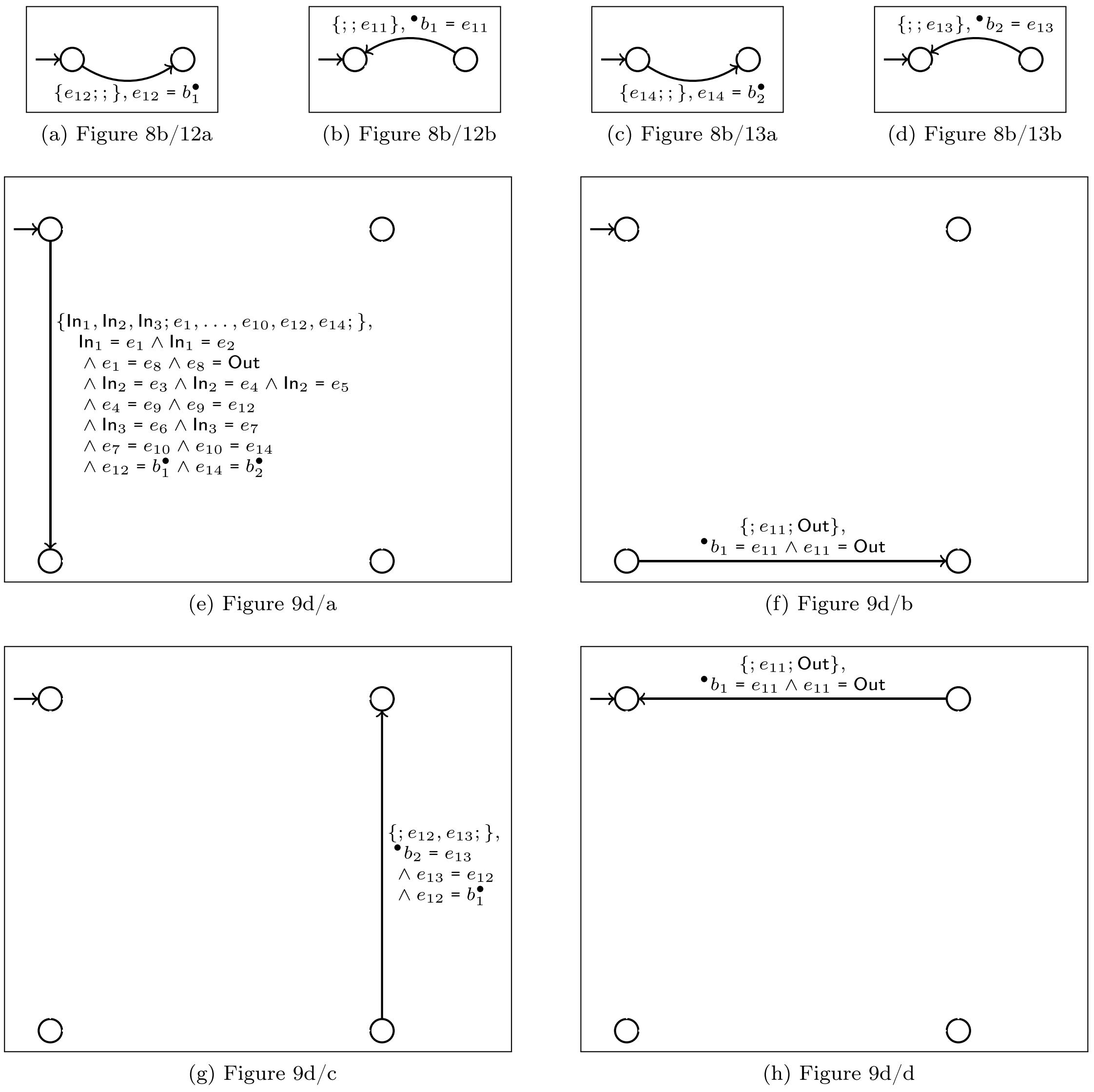

$\{e_{12};;\}, e_{12} = b_1^\bullet$
(a) Figure 8b/12a
$\{;;e_{11}\}, {}^\bullet b_1 = e_{11}$
(b) Figure 8b/12b
$\{e_{14};;\}, e_{14} = b_2^\bullet$
(c) Figure 8b/13a
$\{;;e_{13}\}, {}^\bullet b_2 = e_{13}$
(d) Figure 8b/13b
$\{\mathsf{In}_1, \mathsf{In}_2, \mathsf{In}_3; e_1, \ldots, e_{10}, e_{12}, e_{14};\}$, $\mathsf{In}_1 = e_1 \wedge \mathsf{In}_1 = e_2 \wedge e_1 = e_8 \wedge e_8 = \mathsf{Out} \wedge \mathsf{In}_2 = e_3 \wedge \mathsf{In}_2 = e_4 \wedge \mathsf{In}_2 = e_5 \wedge e_4 = e_9 \wedge e_9 = e_{12} \wedge \mathsf{In}_3 = e_6 \wedge \mathsf{In}_3 = e_7 \wedge e_7 = e_{10} \wedge e_{10} = e_{14} \wedge e_{12} = b_1^\bullet \wedge e_{14} = b_2^\bullet$
(e) Figure 9d/a
$\{;e_{11};\mathsf{Out}\}, {}^\bullet b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
(f) Figure 9d/b
$\{;e_{12}, e_{13};\}, {}^\bullet b_2 = e_{13} \wedge e_{13} = e_{12} \wedge e_{12} = b_1^\bullet$
(g) Figure 9d/c
$\{;e_{11};\mathsf{Out}\}, {}^\bullet b_1 = e_{11} \wedge e_{11} = \mathsf{Out}$
(h) Figure 9d/d

Fig. 13. State-based decomposition example: Alternator$_3$ (cf. Figs. 8b and 9d).

Definition 33. $\bigsqcup \cdot : 2^{\mathbb{A}\text{UTOM}} \to \mathbb{A}\text{UTOM}$ is the function defined as follows:

$$\bigsqcup A = \begin{pmatrix} \bigcup\{\mathsf{Stat}(\alpha) \mid \alpha \in A\}, \\ (\bigcup\{\mathsf{Port}(\alpha) \mid \alpha \in A\}, \bigcup\{\mathsf{Input}(\alpha) \mid \alpha \in A\}, \bigcup\{\mathsf{Outp}(\alpha) \mid \alpha \in A\}), \\ \bigcup\{\mathsf{Memor}(\alpha) \mid \alpha \in A\}, \\ \bigcup\{\mathsf{Trans}(\alpha) \mid \alpha \in A\}, \\ \bigcup\{\mathsf{Init}(\alpha) \mid \alpha \in A\} \end{pmatrix}$$

It is straightforward to see that, in our Alternator$_3$ example, the recomposition of the constraint automata w/MC in Figs. 13e, 13f, 13g, and 13h is equal to the constraint automaton w/MC in Fig. 9d.

The following theorem states that a constraint automaton w/MC equals the recomposition of its state-based decompositions.

Theorem 21 *([35], Lemma 14, p. 32).* $\alpha = \bigsqcup\{\alpha\langle q\rangle \mid q \in \mathsf{Stat}(\alpha)\}$

We are now in a position to formulate what it means for the state-by-state approach to be correct. *Roughly*, the following theorem states that the composition of n local constraint automata w/MC equals the recomposition of that composition's state-based decompositions. More precisely, however, it states that this composition equals the recomposition of *the compo-*

```
{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 $\alpha$ do
    $A := A \cup \{\alpha\}$
    $A' := A' \cup \{\alpha_1\langle q'_1\rangle \otimes \cdots \otimes \alpha_n\langle q'_n\rangle \mid (q, P, \phi, (q'_1,\ldots,q'_n)) \in \mathsf{Trans}(\alpha)\}$
end while
{$\bigsqcup A = \lfloor \alpha_1 \otimes \cdots \otimes \alpha_n \rfloor$}
```

Fig. 14. Algorithm for computing the composition of n constraint automata w/MC using the state-by-state approach. In this figure, we use the same shorthands as those defined below Theorems 19 and 20.

```
{true}
{invar
    [$A' := \{\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle \mid (q_1,\ldots,q_n) \in \mathsf{Init}(\alpha_1) \times \cdots \times \mathsf{Init}(\alpha_n)\}$]
    [$A := \emptyset$]                                                    }
 $A := \emptyset$
 $A' := \{\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle \mid (q_1,\ldots,q_n) \in \mathsf{Init}(\alpha_1) \times \cdots \times \mathsf{Init}(\alpha_n)\}$
{invar}
while $\alpha \in A' \setminus A$ for some $\alpha$ do
    {$\alpha \in A' \setminus A$ and invar and $|\mathsf{Stat}(\alpha_1 \otimes \cdots \otimes \alpha_n)| - |A| = z$}
    {[invar and $0 \leq |\mathsf{Stat}(\alpha_1 \otimes \cdots \otimes \alpha_n)| - |A| < z$]
        [$A' := A' \cup \{\alpha_1\langle q'_1\rangle \otimes \cdots \otimes \alpha_n\langle q'_n\rangle \mid (q, P, \phi, (q'_1,\ldots,q'_n)) \in \mathsf{Trans}(\alpha)\}$]
        [$A := A \cup \{\alpha\}$]                                          }
     $A := A \cup \{\alpha\}$
     $A' := A' \cup \{\alpha_1\langle q'_1\rangle \otimes \cdots \otimes \alpha_n\langle q'_n\rangle \mid (q, P, \phi, (q'_1,\ldots,q'_n)) \in \mathsf{Trans}(\alpha)\}$
    {invar and $0 \leq |\mathsf{Stat}(\alpha_1 \otimes \cdots \otimes \alpha_n)| - |A| < z$}
end while
{invar and [$\alpha \notin A' \setminus A$ for all $\alpha$]}
{$\bigsqcup A = \lfloor \alpha_1 \otimes \cdots \otimes \alpha_n \rfloor$}
```

Fig. 15. Algorithm for computing the composition of n constraint automata w/MC using the state-by-state approach, annotated with assertions for total correctness. In this figure, we use the same shorthands as those defined below Theorems 19 and 20.

sition of state-based decompositions of the local constraint automata w/MC. This is a subtle but important point: it means that to compute the composition of n local constraint automata w/MC, we need to compute only compositions of state-based decompositions of those local constraint automata w/MC. We further clarify this point in the rest of this subsection.

Theorem 22 *([35], Theorem 4, p. 32).*

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

In this theorem, we use the same shorthands as those defined below Theorem 19. Additionally, let $q_1,\ldots,q_n$ denote a sequence of n states.

Together with Theorem 19, the previous theorem establishes that the automaton-by-automaton approach and the state-by-state approach yield weakly behaviorally congruent constraint automata w/MC (and thus compute the same result).

Having formalized de/recomposition, we can now formulate an algorithm for computing the reachable fragment of compositions of n constraint automata w/MC. Fig. 14 shows an algorithm for computing the composition of n local constraint automata w/MC 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 constraint automata w/MC in A). A', in contrast, denotes the full set of so-far computed state-based decompositions (i.e., A' contains A and is such that $A' \setminus A$ contains the constraint automata w/MC 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 constraint automata w/MC in A equals the reachable fragment of the grand composition. Applied to $\mathsf{Alternator}_3$, the algorithm constructs the global constraint automaton w/MC in Fig. 9d in the same order as the order of Figs. 13e, 13f, 13g, and 13h.

Fig. 15 shows the algorithm in Fig. 14 annotated with assertions for total correctness; Fig. 16 shows the loop invariant. This invariant consists of four conjuncts. The first conjunct states that $A \cup A'$ contains the initial states in the final composite. The second conjunct states that the A and A' contain only state-based decompositions of the final composite. The third

$$\begin{array}{ll}
\mathsf{invar}\!: & \{(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle \mid q \in \mathsf{Init}(\alpha_1 \otimes \cdots \otimes \alpha_n)\} \subseteq A \cup (A' \setminus A) \\
& \textbf{and} \;\; A, A' \subseteq \{(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle \mid q \in \mathsf{Stat}(\alpha_1 \otimes \cdots \otimes \alpha_n)\} \\
& \textbf{and} \;\; \left[\left[\begin{array}{c} \alpha \in A \cup (A' \setminus A) \;\; \textbf{implies} \\ \left[\begin{bmatrix} \alpha = (\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q\rangle \\ \textbf{and} \;\; q \in \mathsf{Reach}(\alpha_1 \otimes \cdots \otimes \alpha_n) \end{bmatrix} \;\textbf{for some}\; q\right] \end{array}\right] \;\textbf{for all}\; \alpha\right] \\
& \textbf{and} \;\; \left[\left[\begin{array}{c} [\alpha \in A \;\; \textbf{and} \;\; (q, P, \phi, q') \in \mathsf{Trans}(\alpha)] \;\; \textbf{implies} \\ \left[\begin{bmatrix} \alpha' = (\alpha_1 \otimes \cdots \otimes \alpha_n)\langle q'\rangle \\ \textbf{and} \;\; \alpha' \in A \cup (A' \setminus A) \end{bmatrix} \;\textbf{for some}\; \alpha'\right] \end{array}\right] \;\textbf{for all}\; \alpha, q, q', P, \phi\right]
\end{array}$$

Fig. 16. Addendum to Fig. 15. In this figure, we use the same shorthands as those defined below Theorems 19 and 20.

conjunct states that every constraint automaton w/MC in $A \cup A'$ is a state-based decomposition of the final composite, with respect to some reachable state in that final composite. The fourth conjunct states that if a constraint automaton w/MC in A has a transition entering a (global) state q', $A \cup A'$ contains a decomposition of the final composite with respect to q'. As soon as the loop terminates, the invariant and the negated loop condition imply that every constraint automaton w/MC in A has a reachable significant state ("soundness"; consequence of the third conjunct) and that, in fact, A contains a constraint automaton w/MC for every reachable state ("completeness"; consequence of the fourth conjunct).

Theorem 23 *([35], Theorem 6, p. 35). The algorithm in Fig. 14 is correct.*

Note that the invariant refers only to decompositions of the global constraint automaton w/MC 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 constraint automata w/MC 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 constraint automata w/MC, the algorithm never needs to compute any intermediate compounds, so avoiding a potential source of exponential resource requirements.

7.2. Experimental results and analysis

We implemented the state-by-state approach to composition as an extension to our Reo-to-Java compiler. To evaluate the performance of the state-by-state approach, we experimented with the same k-parametric families of connectors as those in Fig. 11 under the same experimental conditions as those described in Section 6.2. Fig. 17 shows the composition times that we measured for both the automaton-by-automaton approach (dotted blue lines, same as in Fig. 11) and the state-by-state approach (solid yellow lines, new), for all connectors, and for several values of $2 \le k \le 64$.

The four families whose compositions grow exponentially in k (i.e., $\mathsf{EarlyAsyncBarrierMerger}_k$, $\mathsf{EarlyAsyncMerger}_k$, $\mathsf{LateAsyncReplicator}_k$, and $\mathsf{LateAsyncRouter}_k$) logically provoke exponential growth in resource requirements not only in the automaton-by-automaton approach (as already observed in Section 6.2) but also in the state-by-state approach.

For $\mathsf{EarlyAsyncOutSequencer}_k$ and Lock_k, as shown in Figs. 17d and 17i, the state-by-state approach has substantially better performance than the automaton-by-automaton approach: whereas the automaton-by-automaton approach fails for $k > 14$, the state-by-state approach succeeds for all values of k under study. This is explained by the fact that the *reachable* state spaces of these families (which the state-by-state approach computes) grow linearly in k, whereas their *complete* state spaces (which the automaton-by-automaton approach computes) grow exponentially in k until cycles are closed. (Recall that these two families formed the main motivation for developing the state-by-state approach.)

For $\mathsf{EarlyAsyncBarrierMerger}_k$ and $\mathsf{EarlyAsyncMerger}_k$, $\mathsf{EarlyAsyncReplicator}_k$ and $\mathsf{LateAsyncMerger}_k$, the state-by-state approach seems substantially slower than the automaton-by-automaton approach. A mundane reason may be that we have not optimized our implementation of the state-by-state approach as aggressively as our implementation of the automaton-by-automaton 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 automaton-by-automaton approach, because the state-by-state approach continuously accesses all local constraint automata w/MC.

Finally, $\mathsf{Alternator}_k$ clearly forms a problematic case for the state-by-state approach. For this family, the automaton-by-automaton approach performs much better. An analysis of the result of the state-by-state composition of $\mathsf{Alternator}_k$ tells us that a large part of the state space becomes unreachable when we subtract internal ports, aggregate internal transitions and prune unreachable states; refer to Fig. 18 for a sketch of the state space for $k = 4$. This unreachable-by-aggregation part of the state space of the result is "useless" as far as the behavior of the composition is concerned.

An example of such a state for $\mathsf{Alternator}_k$ is the state signifying that all buffers are full, except for the buffer closest to the output port. When using the automaton-by-automaton approach, this state is "bypassed" early on (as a result of aggregation) by having all of its incoming transitions rerouted to the state in which all buffers are full, except the buffer

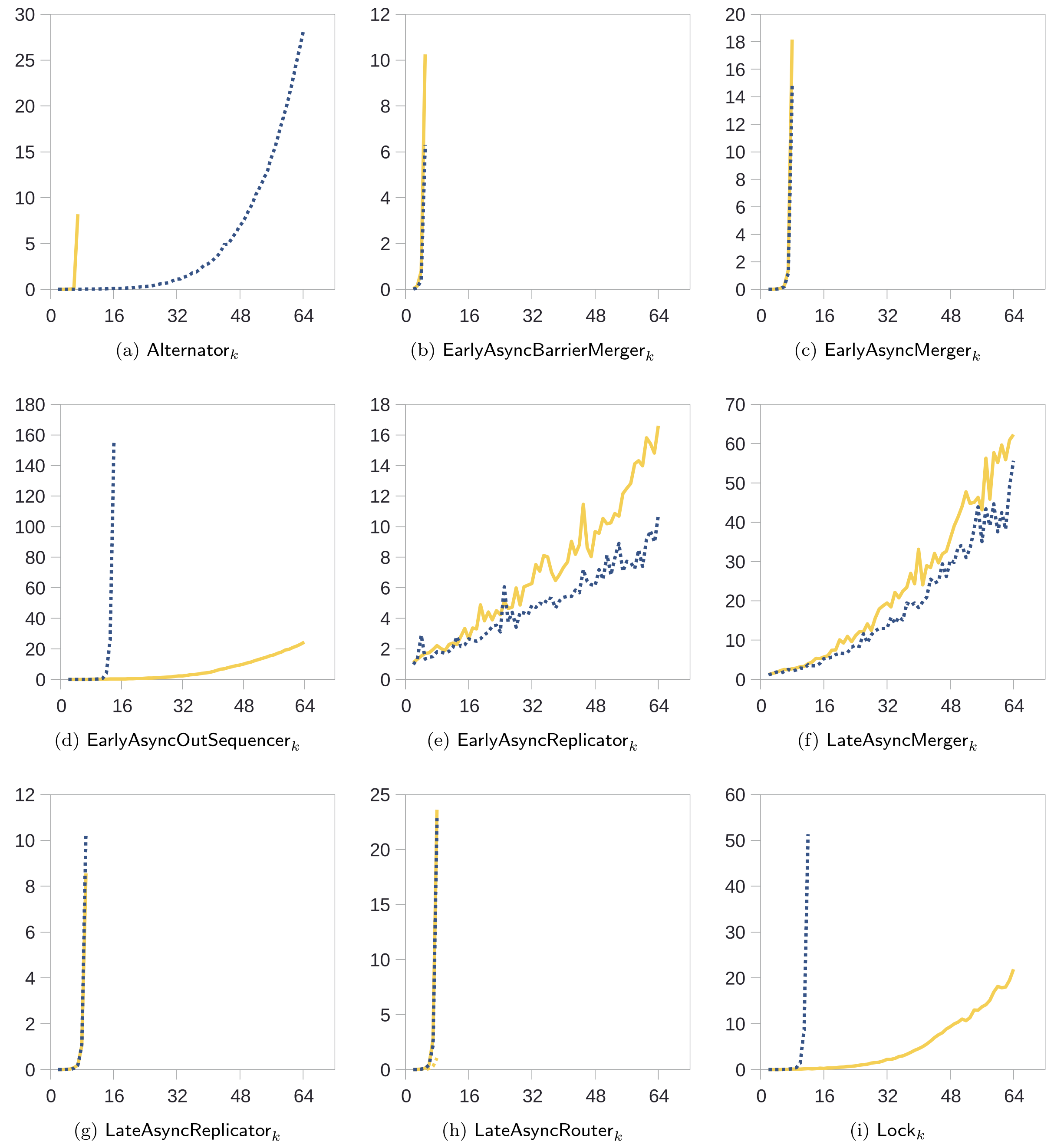


Fig. 17. Composition times (y-axis) for nine k-parametric families, for $2 \le k \le 64$ (x-axis), by applying the automaton-by-automaton approach (dotted blue lines) and the state-by-state approach after subtraction/aggregation (solid yellow lines). Time is measured in seconds, except for EarlyAsyncReplicator$_k$ and LateAsyncMerger$_k$, where time is measured in milliseconds. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

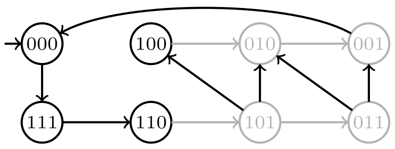


Fig. 18. Result of the state-by-state approach when applied to Alternator$_4$. State labels xyz indicate the emptiness/fullness of buffers, where z refers to the buffer closest to the output, y to the middle buffer and z to the buffer furthest from the output. Internal transitions are shown in gray, as are states that are unreachable-by-aggregation or unreachable-by-transitivity.

furthest from the output port. The automaton-by-automaton approach can then safely prune the state from its intermediate result. In contrast, because the state-by-state approach performs neither subtraction nor aggregation on the intermediate results, it cannot prune these states as they appear.

While it is certainly possible to evaluate $\bigsqcup A$ in each iteration of the state-by-state approach and perform subtraction of the appropriate ports as well as (some form of) aggregation, it would be counterproductive to prune states: a state q that is unreachable in an intermediate result may yet become reachable in a later iteration, if we expand a state q' that turns out to have a transition reaching q. We therefore surmise that, like the automaton-by-automaton approach, the state-by-state approach may also compute an unnecessary part of the state space, namely the part that is unreachable-by-aggregation.

Interestingly, early subtraction and aggregation by themselves do 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 beyond the scope of this paper [36]. Here, the important point is that, even though the 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; i.e., when unreachability-by-multiplication dominates unreachability-by-aggregation and vice versa.

Another piece of future work concerns the investigation of a variant of the state-by-state approach with early subtraction/aggregation/pruning similar to the automaton-by-automaton approach. The main challenge with this is that to prune, we require certain information that, in the state-by-state approach, seems to become available only after we have completed computing the final composite; we need to develop clever techniques to obtain this kind of information earlier in the process. For instance, it seems useful to have a sufficient condition for a state to be unreachable in the final result, thus allowing us to prune it in an early iteration and skip its expansion.

7.3. Related work

The main inspiration for the state-by-state approach for composing constraint automata W/MC came from Proença's distributed Reo engine [37]. On input of a connector, this engine starts an actor for each of that connector's nodes and channels. Each of these actors maintains a local automaton (not quite a constraint automaton W/MC but the differences and details do not matter here) for its corresponding node or 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 compositions effectively does a similar computation at compile-time.

Some literature exists on algorithms for composing constraint automata (without memory cells). For instance, Ghassemi et al. documented that the order in which a tool composes n constraint automata (without memory cells) matters [38]: although any order yields the same final composite (because multiplication is associative and commutative by Theorems 5 and 6), different orders may yield different intermediate composites. Some orders may give rise to relatively large intermediate composites, with high resource requirements as a result, while other orders may keep intermediate composites 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* constraint automata (without memory cells) in a state-by-state approach, but they do not generalize this to arbitrary compositions as we do in this paper. Pourvatan and Rouhy also worked on an algorithm for composing constraint automata (without memory cells) [39]. Their approach consists of a special algebraic representation of constraint automata (without memory cells), including a reformulation of multiplication to support this representation. Pourvatan and Rouhy claim that their approach computes composition twice as fast as the approach by Ghassemi et al., but they provide only limited empirical evidence.

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 [40]. The fact that we need to compose constraint automata W/MC during the expansion of global states—and explicitly do not want to compute the composition beforehand—makes our situation more complex, though. Theorem 20 plays a key role in this respect.

Our work is related also to on-the-fly model checking, proposed by Gerth et al. [41], where the state space under verification is generated as needed during the actual decision procedure instead of in its entirety. If a counterexample is found already early during state space generation/exploration, then, no effort gets wasted on precomputing the entire state space. A technical 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. More conceptually, the main difference between our approach and on-the-fly model checking is that an on-the-fly model checker can stop generating the state space as soon as a counterexample is found, whereas we always need to compute the entire reachable state space.

8. Conclusion

In Sections 3, 4, and 5, we presented an extension of constraint automata with a mechanism to finitely and compactly deal with infinite data domains, based on memory cells, including definitions of foundational notions as languages, language equivalence, bisimilarity, and operations for composition. Although ad-hoc presentations of constraint automata with memory cells have appeared in previous work [9–11], a rigorous and formal presentation as comprehensive as ours in this paper does not yet exist.

In Sections 6 and 7, we subsequently studied—including practical experiments—two approaches for computing the composition of n constraint automata with memory cells: the automaton-by-automaton approach and the state-by-state approach. The former approach is arguably more natural; the latter approach solves a problem of the former approach where intermediate composites grow exponentially while their final composite grows only linearly. We proved the correctness of the algorithm in the state-by-state approach using Hoare logic. Our experimental results show that the state-by-state approach substantially improves the problematic cases of the automaton-by-automaton approach. However, in other cases, the automaton-by-automaton approach outperforms the state-by-state approach. In future work, we want to investigate heuristics for deciding which of these two approaches we should use when. We also want to study whether we can combine the strong points of the automaton-by-automaton approach (early subtraction/aggregation) with the strong points of the state-by-state approach (only reachable states are computed). Another important piece of future work is investigating the extent to which the state-by-state approach is compatible with other compiler optimization techniques that we developed in previous work [6,5,7].

Our primary use case for the development of the state-by-state approach was compilation of connectors to executable code. In our compiler, we use constraint automata to compositionally represent connector behavior, and our compiler therefore needs to compose small constraint automata into larger ones as efficiently as possible. When we observed unprecedented excessive compilation times for certain connectors, we realized that there was a fundamental issue with the automaton-by-automaton approach to composition. The state-by-state approach resolves these issues, as shown in our experiments in Sections 6 and 7. The implementations of the automaton-by-automaton approach and the state-by-state approach that we used in these experiments are part of our compiler. The state-by-state approach has, however, wider applicability, beyond compilation. Also in constraint automaton model checking [23–26], for instance, the reachable state space of a constraint automaton needs to be computed; the state-by-state approach helps here, too. Yet another application is connector animation [42]. We have an animation framework for Reo (part of the Reo toolset; reo.project.cwi.nl) that allows users to graphically animate data-flows through connectors; under the hood, also this animation framework needs to compute the reachable state space.

Although inspired by Reo, constraint automata are a generic operational formalism suitable for compositional specification of concurrent systems, either directly, or as a semantic model for other languages/models. For instance, the semantics of the actor-based modeling language Rebeca has been defined in terms of constraint automata (without memory cells) [43,44]. From our perspective, Reo is just one convenient syntax for writing multiplication/subtraction/aggregation expressions of constraint automata. Different syntax alternatives for constraint automata may work equally well or yield perhaps even more user-friendly languages. For instance, we can translate UML sequence/activity diagrams and BPMN to constraint automata [45–47]. Algebras of Bliudze and Sifakis [19,20], originally developed for BIP [48], also have a straightforward interpretation in terms of constraint automata [49,50], thereby offering an interesting alternative possible syntax. Due to their generality, constraint automata 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 work presented in this paper applicable also beyond Reo.

Appendix A. Proofs for Section 4

Proof of Theorem 1. By applying Definition 17, conclude that $\approx$ is reflexive, symmetric, and transitive. □

Proof of Theorem 2. For some α, let $R = \{(q, q) \mid q \in \mathsf{Stat}(\alpha)\}$. Then, by applying Definition 18, conclude $\alpha \preceq_R \alpha$. Then, by applying Definition 19, conclude $\alpha \preceq \alpha$. Thus, conclude that $\preceq$ is reflexive.

For some $\alpha_1, \alpha_2, \alpha_3$, suppose $\left[\alpha_1 \preceq \alpha_2 \textbf{ and } \alpha_2 \preceq \alpha_3\right]$. Then, by applying Definition 19, for some R_1, R_2, conclude $\left[\alpha_1 \preceq_{R_1} \alpha_2 \textbf{ and } \alpha_2 \preceq_{R_2} \alpha_3\right]$. Let $R = \{(q_1, q_3) \mid q_1\ R_1\ q_2\ R_2\ q_3\}$. Then, by applying Definition 18, conclude $\alpha_1 \preceq_R \alpha_3$. Then, by applying Definition 19, conclude $\alpha_1 \preceq \alpha_3$. Thus, conclude that $\preceq$ is transitive. □

Proof of Theorem 3. By applying Definitions 20 and 21, and by applying (the definitions of R in the proof of) Theorem 2, $\simeq$ is reflexive and transitive. By applying Definitions 20 and 21, $\simeq$ is symmetric. □

Proof of Theorem 4. For some α_1, α_2, suppose $\left[\alpha_1 \simeq \alpha_2 \textbf{ and } w \in \mathsf{Behav}(\alpha_1)\right]$. Then, by applying Definition 16, for some q_1, $q_1', \ldots, w', w'', \ldots, \mu_1, \mu_1', \ldots$, conclude:

$$q_1 \in \mathsf{Init}(\alpha_1) \textbf{ and } (q_1, w, \mu_1) \vdash_{\alpha_1} (q_1', w', \mu_1') \vdash_{\alpha_1} \cdots$$

Then, because $\alpha_1 \simeq \alpha_2$ and by applying Definitions 18, 20, and 21, for some q_2 and R, conclude:

$$q_2 \in \mathsf{Init}(\alpha_2) \textbf{ and } q_1\ R\ q_2$$

For some $\hat{q}_1, \hat{q}_2, \hat{w}, \hat{w}', \hat{\mu}, \hat{\mu}'$, suppose $\left[(\hat{q}_1, \hat{w}, \hat{\mu}) \vdash_{\alpha_1} (\hat{q}_1', \hat{w}', \hat{\mu}') \textbf{ and } \hat{q}_1\ R\ \hat{q}_2\right]$. Then, by applying Definition 15, for some P, ϕ_1, λ, distinguish two cases.

- Case:

$$\begin{aligned}&(\hat{q}_1, P, \phi_1, \hat{q}_1') \in \mathsf{Trans}(\alpha_1)\\ &\textbf{and } \mathsf{Dom}(\hat{\mu}) = \mathsf{Dom}(\hat{\mu}') = \mathsf{Memor}(\alpha_1) \textbf{ and } \mathsf{Dom}(\lambda) = P \neq \emptyset\\ &\textbf{and } \lambda \cup \{{}^\bullet m \mapsto \mu(m) \mid m \in \mathsf{Memor}(\alpha_1)\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in \mathsf{Memor}(\alpha_1)\} \models \phi_1\\ &\textbf{and } \hat{w}' = \lambda \hat{w}\end{aligned}$$

  Then, because $\alpha_1 \simeq \alpha_2$, and because $\hat{q}_1 \; R \; \hat{q}_2$, and by applying Definitions 18, 20, and 21, for some $\hat{q}_2', \phi_2$, conclude:

$$\begin{aligned}&(\hat{q}_2, P, \phi_2, \hat{q}_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } \hat{q}_1' \; R \; \hat{q}_2'\\ &\textbf{and } \mathsf{Dom}(\hat{\mu}) = \mathsf{Dom}(\hat{\mu}') = \mathsf{Memor}(\alpha_2)\\ &\textbf{and } \lambda \cup \{{}^\bullet m \mapsto \mu(m) \mid m \in \mathsf{Memor}(\alpha_2)\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in \mathsf{Memor}(\alpha_2)\} \models \phi_2\end{aligned}$$

  Then, because $\mathsf{Dom}(\lambda) = P \neq \emptyset$, by applying Definition 15, conclude $(\hat{q}_2, \hat{w}, \hat{\mu}) \vdash_{\alpha_2} (\hat{q}_2', \hat{w}', \hat{\mu}')$.
- Case:

$$\begin{aligned}&(\hat{q}_1, \emptyset, \phi_1, \hat{q}_1') \in \mathsf{Trans}(\alpha_1)\\ &\textbf{and } \mathsf{Dom}(\hat{\mu}) = \mathsf{Dom}(\hat{\mu}') = \mathsf{Memor}(\alpha_1)\\ &\textbf{and } \{{}^\bullet m \mapsto \mu(m) \mid m \in \mathsf{Memor}(\alpha_1)\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in \mathsf{Memor}(\alpha_1)\} \models \phi_1\\ &\textbf{and } \hat{w}' = \hat{w}\end{aligned}$$

  Then, because $\alpha_1 \simeq \alpha_2$, and because $\hat{q}_1 \; R \; \hat{q}_2$, and by applying Definitions 18, 20, and 21, for some $\hat{q}_2', \phi_2$, conclude:

$$\begin{aligned}&(\hat{q}_2, \emptyset, \phi_2, \hat{q}_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } \hat{q}_1' \; R \; \hat{q}_2'\\ &\textbf{and } \mathsf{Dom}(\hat{\mu}) = \mathsf{Dom}(\hat{\mu}') = \mathsf{Memor}(\alpha_2)\\ &\textbf{and } \{{}^\bullet m \mapsto \mu(m) \mid m \in \mathsf{Memor}(\alpha_2)\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in \mathsf{Memor}(\alpha_2)\} \models \phi_2\end{aligned}$$

  Then, by applying Definition 15, conclude $(\hat{q}_2, \hat{w}, \hat{\mu}) \vdash_{\alpha_2} (\hat{q}_2', \hat{w}', \hat{\mu}')$.

Thus, for some $\hat{q}_2$, conclude $\left[(\hat{q}_2, \hat{w}, \hat{\mu}) \vdash_{\alpha_2} (\hat{q}_2', \hat{w}', \hat{\mu}') \textbf{ and } \hat{q}_1' \; R \; \hat{q}_2'\right]$.

Thus, for all $\hat{q}_1, \hat{q}_2, \hat{w}, \hat{w}', \hat{\mu}, \hat{\mu}'$, conclude:

$$\begin{aligned}&\left[(\hat{q}_1, \hat{w}, \hat{\mu}) \vdash_{\alpha_1} (\hat{q}_1', \hat{w}', \hat{\mu}') \textbf{ and } \hat{q}_1 \; R \; \hat{q}_2\right] \textbf{ implies}\\ &\left[\left[(\hat{q}_2, \hat{w}, \hat{\mu}) \vdash_{\alpha_2} (\hat{q}_2', \hat{w}', \hat{\mu}') \textbf{ and } \hat{q}_1' \; R \; \hat{q}_2'\right] \textbf{ for some } \hat{q}_2'\right]\end{aligned}$$

Then, because $(q_1, w, \mu_1) \vdash_{\alpha_1} (q_1', w', \mu_1') \vdash_{\alpha_1} \cdots$, and because $q_1 \; R \; q_2$, and by applying induction, conclude $(q_2, w, \mu_2) \vdash_{\alpha_2} (q_2', w', \mu_2') \vdash_{\alpha_2} \cdots$. Then, because $q_2 \in \mathsf{Init}(\alpha_2)$, and by applying Definition 16, conclude $w \in \mathsf{Behav}(\alpha_2)$. Then, conclude $\mathsf{Behav}(\alpha_1) \subseteq \mathsf{Behav}(\alpha_2)$. Symmetrically, conclude:

$$\mathsf{Behav}(\alpha_2) \subseteq \mathsf{Behav}(\alpha_1)$$

Then, by applying Definition 17, conclude $\alpha_1 \approx \alpha_2$.

Thus, conclude $\alpha_1 \simeq \alpha_2$ implies $\alpha_1 \approx \alpha_2$. Then, conclude $\simeq \subseteq \approx$. □

Appendix B. Proofs for Section 5

Proof of Theorem 5. For some α_1, α_2, let $R = \{((q_1, q_2), (q_2, q_1)) \mid q_1 \in \mathsf{Stat}(\alpha_1) \textbf{ and } q_2 \in \mathsf{Stat}(\alpha_2)\}$. Then, by applying Definition 20, conclude $\alpha_1 \otimes \alpha_2 \simeq_R \alpha_2 \otimes \alpha_1$. Then, by applying Definition 21, conclude $\alpha_1 \otimes \alpha_2 \simeq \alpha_2 \otimes \alpha_1$. □

Proof of Theorem 6. For some $\alpha_1, \alpha_2, \alpha_3$, let:

$$R = \{((q_1, (q_2, q_2)), ((q_1, q_2), q_3)) \mid q_1 \in \mathsf{Stat}(\alpha_1) \textbf{ and } q_2 \in \mathsf{Stat}(\alpha_2) \textbf{ and } q_3 \in \mathsf{Stat}(\alpha_3)\}$$

Then, by applying Definition 20, conclude $\alpha_1 \otimes (\alpha_2 \otimes \alpha_3) \simeq_R (\alpha_1 \otimes \alpha_2) \otimes \alpha_3$. Then, by applying Definition 21, conclude $\alpha_1 \otimes (\alpha_2 \otimes \alpha_3) \simeq (\alpha_1 \otimes \alpha_2) \otimes \alpha_3$. □

Proof of Theorem 7. For some $\alpha_1, \alpha_2, \alpha_3, \alpha_4$, suppose $\left[\alpha_1 \simeq \alpha_2 \textbf{ and } \alpha_3 \simeq \alpha_4\right]$. Then, by applying Definition 21, for some R_{12}, R_{34}, conclude $\left[\alpha_1 \simeq_{R_{12}} \alpha_2 \textbf{ and } \alpha_3 \simeq_{R_{34}} \alpha_4\right]$.

Let $R = \{(q_1, q_3) \; R \; (q_2, q_4) \mid q_1 \; R_{12} \; q_2 \textbf{ and } q_3 \; R_{34} \; q_4\}$.

For some $q_{13}, q_{13}', q_{24}, P_{13}, \phi_{13}$, suppose $\left[(q_{13}, P_{13}, \phi_{13}, q_{13}') \in \mathsf{Trans}(\alpha_1 \otimes \alpha_3) \textbf{ and } q_{13} \; R \; q_{24}\right]$. Then by applying Definition 22, for some $q_1, q_1', q_3, q_3', P_1, P_3, \phi_1, \phi_3$, conclude:

$$\begin{aligned}&(q_1, P_1, \phi_1, q_1') \in \mathsf{Trans}(\alpha_1) \textbf{ and } (q_3, P_3, \phi_3, q_3') \in \mathsf{Trans}(\alpha_3) \textbf{ and } \mathsf{Port}(\alpha_1) \cap P_3 = \mathsf{Port}(\alpha_3) \cap P_1\\ &\textbf{and } P_{13} = P_1 \cup P_3 \textbf{ and } \phi_{13} = \phi_1 \wedge \phi_3 \textbf{ and } q_{13}' = (q_1', q_3')\end{aligned}$$

Then, because $\alpha_1 \simeq_{R_{12}} \alpha_3$, and because $\alpha_2 \simeq_{R_{24}} \alpha_4$, and because $q_{13}\ R\ q_{24}$, and by applying Definitions 18, 20, and 21, for some q_2, q_4, conclude:

$$\begin{array}{l}\phi_1 \Rightarrow \bigvee\{\phi_2 \mid (q_2, P_1, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_1'\ R_{12}\ q_2'\}\\ \textbf{and } \phi_2 \Rightarrow \bigvee\{\phi_4 \mid (q_4, P_3, \phi_4, q_4') \in \mathsf{Trans}(\alpha_4) \textbf{ and } q_3'\ R_{34}\ q_4'\}\\ \textbf{and } \mathsf{Port}(\alpha_2) \cap P_3 = \mathsf{Port}(\alpha_4) \cap P_1 \textbf{ and } q_{24} = (q_2, q_4)\end{array}$$

Then, conclude:

$$\phi_1 \wedge \phi_3 \Rightarrow \bigvee \left\{ \phi_2 \wedge \phi_4 \;\middle|\; \begin{array}{l}(q_2, P_1, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } (q_4, P_3, \phi_4, q_4') \in \mathsf{Trans}(\alpha_4)\\ \textbf{and } q_1'\ R_{12}\ q_2' \textbf{ and } q_3'\ R_{34}\ q_4'\end{array} \right\}$$

Then, because $q_{13}' = (q_1', q_3')$, and because $q_{24} = (q_2, q_4)$, and because $P_{13} = P_1 \cup P_3$, and because $\phi_{13} = \phi_1 \wedge \phi_3$, and because $\mathsf{Port}(\alpha_2) \cap P_3 = \mathsf{Port}(\alpha_4) \cap P_1$, and by the definition of R, conclude:

$$\phi_{13} \Rightarrow \bigvee\{\phi_2 \wedge \phi_4 \mid (q_{24}, P_{13}, \phi_2 \wedge \phi_4, (q_2', q_4')) \in \mathsf{Trans}(\alpha_2 \otimes \alpha_4) \textbf{ and } q_{13}'\ R\ (q_2', q_4')\}$$

Then, by applying Definition 18, conclude $\alpha_1 \otimes \alpha_3 \preceq_R \alpha_2 \otimes \alpha_4$. Symmetrically, conclude $\alpha_2 \otimes \alpha_4 \preceq_{R^{-1}} \alpha_1 \otimes \alpha_3$. Then, by applying Definitions 20 and 21, conclude $\alpha_1 \otimes \alpha_3 \simeq \alpha_2 \simeq \alpha_4$. □

Proof of Theorem 8. For some α_1, α_2, suppose $\alpha_1 \simeq \alpha_2$. Then, by applying Definition 21, for some R, conclude $\alpha_1 \simeq_R \alpha_2$.

For some $q_1, q_1', q_2, P_\ominus, \phi_\ominus$, suppose $\left[(q_1, P_\ominus, \phi_\ominus, q_1') \in \mathsf{Trans}(\alpha_1 \ominus P) \textbf{ and } q_1\ R\ q_2\right]$. Then, by applying Definitions 23 and 24, conclude $\left[(q_1, P_1, \phi_1, q_1') \in \mathsf{Trans}(\alpha_1) \textbf{ and } P_\ominus = P_1 \setminus P \textbf{ and } \phi_\ominus = \exists P.\phi_1\right]$. Then, because $\alpha_1 \simeq_R \alpha_2$, and because $q_1\ R\ q_2$, and by applying Definitions 18, 20, and 21, conclude $\phi_1 \Rightarrow \bigvee\{\phi_2 \mid (q_2, P_1, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_1'\ R\ q_2'\}$. Then, conclude:

$$\exists P.\phi_1 \Rightarrow \bigvee\{\exists P.\phi_2 \mid (q_2, P_1, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_1'\ R\ q_2'\}$$

Then, because $P_\ominus = P_1 \setminus P$, and because $\phi_\ominus = \exists P.\phi_1$, and by applying Definitions 23 and 24, conclude $\phi_\ominus \Rightarrow \bigvee\{\exists P.\phi_2 \mid (q_2, P_\ominus, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2 \ominus P) \textbf{ and } q_1'\ R\ q_2'\}$. Then, by applying Definition 18, conclude $\alpha_1 \ominus P \preceq_R \alpha_2 \ominus P$. Then, symmetrically, conclude $\alpha_2 \ominus P \preceq_{R^{-1}} \alpha_1 \ominus P$. Then, by applying Definitions 20 and 21, conclude $\alpha_1 \ominus P \simeq \alpha_2 \ominus P$. □

Proof of Theorem 9. For some P, α_1, α_2, suppose $P \cap \mathsf{Port}(\alpha_1) \cap \mathsf{Port}(\alpha_2) = \emptyset$.

Let $R = \{((q_1, q_2), (q_1, q_2)) \mid q_1 \in \mathsf{Stat}(\alpha_1) \textbf{ and } q_2 \in \mathsf{Stat}(\alpha_2)\}$.

For some $q_{12\ominus}, q_{12\ominus}', q_{1\ominus2\ominus}, P_{12\ominus}, \phi_{12\ominus}$, suppose:

$$(q_{12\ominus}, P_{12\ominus}, \phi_{12\ominus}, q_{12\ominus}') \in \mathsf{Trans}((\alpha_1 \otimes \alpha_2) \ominus P) \textbf{ and } q_{12\ominus}\ R\ q_{1\ominus2\ominus}$$

Then, by applying Definitions 22, 23, and 24, for some $q_1, q_1', q_2, q_2', P_1, P_2, \phi_1, \phi_2$, conclude:

$$\begin{array}{l}(q_1, P_1, \phi_1, q_1') \in \mathsf{Trans}(\alpha_1) \textbf{ and } (q_2, P_2, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2) \textbf{ and } \mathsf{Port}(\alpha_1) \cap P_2 = \mathsf{Port}(\alpha_2) \cap P_1\\ \textbf{and } q_{12\ominus} = (q_1, q_2) \textbf{ and } q_{12\ominus}' = (q_1', q_2') \textbf{ and } P_{12\ominus} = (P_1 \cup P_2) \setminus P \textbf{ and } \phi_{12\ominus} = \exists P.(\phi_1 \wedge \phi_2)\end{array}$$

Then, conclude $\mathsf{Port}(\alpha_1) \cap (P_2 \setminus P) = \mathsf{Port}(\alpha_2) \cap (P_1 \setminus P)$. Then, because $(q_1, P_1, \phi_1, q_1') \in \mathsf{Trans}(\alpha_1)$, and because $(q_2, P_2, \phi_2, q_2') \in \mathsf{Trans}(\alpha_2)$, and by Definitions 22, 23, and 24, conclude:

$$((q_1, q_2), (P_1 \setminus P) \cup (P_2 \setminus P), (\exists P.\phi_1) \wedge (\exists P.\phi_2), (q_1', q_2')) \in \mathsf{Trans}((\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P))$$

Then, because $q_{12\ominus}\ R\ q_{1\ominus2\ominus}$, and because $q_{12\ominus} = (q_1, q_2)$, and because $q_{12\ominus}' = (q_1', q_2')$, and by the definition of R, conclude:

$$(q_{1\ominus2\ominus}, (P_1 \setminus P) \cup (P_2 \setminus P), (\exists P.\phi_1) \wedge (\exists P.\phi_2), (q_1', q_2')) \in \mathsf{Trans}((\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)) \textbf{ and } q_{12\ominus}'\ R\ (q_1', q_2')$$

Then, because $(P_1 \cup P_2) \setminus P = (P_1 \setminus P) \cup (P_2 \setminus P)$, and because $P_{12\ominus} = (P_1 \cup P_2) \setminus P$, conclude:

$$(q_{1\ominus2\ominus}, P_{12\ominus}, (\exists P.\phi_1) \wedge (\exists P.\phi_2), (q_1', q_2')) \in \mathsf{Trans}((\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)) \textbf{ and } q_{12\ominus}'\ R\ (q_1', q_2')$$

Then, because $\exists P.(\phi_1 \wedge \phi_2) \Rightarrow (\exists P.\phi_1) \wedge (\exists P.\phi_2)$, and because $\phi_{12\ominus} = \exists P.(\phi_1 \wedge \phi_2)$, conclude:

$$\phi_{12\ominus} \Rightarrow \bigvee\{\hat{\phi}_{1\ominus2\ominus} \mid (\hat{q}_{1\ominus2\ominus}, P_{12\ominus}, \hat{\phi}_{1\ominus2\ominus}, (\hat{q}_1', \hat{q}_2')) \in \mathsf{Trans}((\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)) \textbf{ and } q_{12\ominus}'\ R\ (\hat{q}_1', \hat{q}_2')\}$$

Then, by applying Definition 18, conclude $(\alpha_1 \otimes \alpha_2) \ominus P \preceq_R (\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)$.

The proof that $(\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P) \preceq_{R^{-1}} (\alpha_1 \otimes \alpha_2) \ominus P$ is analogous, except for two steps: generally, $\mathsf{Port}(\alpha_1) \cap (P_2 \setminus P) = \mathsf{Port}(\alpha_2) \cap (P_1 \setminus P)$ does not imply $\mathsf{Port}(\alpha_1) \cap P_2 = \mathsf{Port}(\alpha_2) \cap P_1$, and generally, $(\exists P.\phi_1) \wedge (\exists P.\phi_2) \not\Rightarrow \exists P.(\phi_1 \wedge \phi_2)$. Because $P \cap \mathsf{Port}(\alpha_1) \cap \mathsf{Port}(\alpha_2) = \emptyset$, and because $P_1 \subseteq \mathsf{Port}(\alpha_1)$ because $P_2 \subseteq \mathsf{Port}(\alpha_2)$, conclude:

$$\begin{array}{l}\mathsf{Port}(\alpha_1) \cap P_2 = (\mathsf{Port}(\alpha_1) \cap P_2) \setminus P = \mathsf{Port}(\alpha_1) \cap (P_2 \setminus P) = \mathsf{Port}(\alpha_2) \cap (P_1 \setminus P) = (\mathsf{Port}(\alpha_2) \cap P_1) \setminus P\\ \qquad\qquad = \mathsf{Port}(\alpha_2) \cap P_1\end{array}$$

Similarly, because $P \cap \mathsf{Port}(\alpha_1) \cap \mathsf{Port}(\alpha_2) = \emptyset$, and because $P_1 \subseteq \mathsf{Port}(\alpha_1)$ because $P_2 \subseteq \mathsf{Port}(\alpha_2)$, and because $\mathsf{Free}(\phi_1) \subseteq P_1 \cup {}^\bullet M_1 \cup M_1^\bullet$, and because $\mathsf{Free}(\phi_2) \subseteq P_2 \cup {}^\bullet M_2 \cup M_2^\bullet$, conclude:

$$(\exists P.\phi_1) \wedge (\exists P.\phi_2) \not\Rightarrow \exists P.(\phi_1 \wedge \phi_2)$$

Thus, conclude $(\alpha_1 \otimes \alpha_2) \ominus P \preceq_R (\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)$ and $(\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P) \preceq_{R^{-1}} (\alpha_1 \otimes \alpha_2) \ominus P$. Then, by applying Definitions 20 and 21, conclude $(\alpha_1 \otimes \alpha_2) \ominus P \simeq (\alpha_1 \ominus P) \otimes (\alpha_2 \ominus P)$. □

Proof of Theorem 10. For some α_1, α_2, suppose $\alpha_1 \simeq \alpha_2$. Then, by applying Definition 21, for some R, conclude $\alpha_1 \simeq_R \alpha_2$.

For some $q_1, q_1', q_2, P_{1\star}, \phi_{1\star}$, suppose $\left[(q_1, P_{1\star}, \phi_{1\star}, q_1') \in \mathsf{Trans}(\alpha^\star) \textbf{ and } q_1 \; R \; q_2\right]$. Then, by applying Definition 27, for some $n, q_{1,1}, \ldots, q_{1,n+1}, \phi_{1,1}, \ldots, \phi_{1,n}$, distinguish two cases:

- Case:

$$\begin{aligned}
&(q_{1,1}, \emptyset, \phi_{1,1}, q_{1,2}), \ldots, (q_{1,n-1}, \emptyset, \phi_{1,n-1}, q_{1,n}), (q_{1,n}, P_{1\star}, \phi_{1,n}, q_{1,n+1}) \in \mathsf{Trans}(\alpha_1)\\
&\textbf{and } q_1 = q_{1,1} \textbf{ and } q_1' = q_{1,n+1} \textbf{ and } P_{1\star} \neq \emptyset\\
&\textbf{and } \phi_{1\star} = \exists \mathsf{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))
\end{aligned}$$

Then, because $\alpha_1 \simeq \alpha_2$, and because $q_1 \; R \; q_2$, and by applying Definitions 18, 20, and 21, and by applying induction, for some $q_{2,1}, \ldots, q_{2,n+1}$, conclude:

$$\begin{aligned}
&\phi_{1,1} \Rightarrow \bigvee\{\phi_{2,1} \mid (q_{2,1}, \emptyset, \phi_{2,1}, q_{2,2}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,2} \; R \; q_{2,2}\}\\
&\textbf{and } \cdots\\
&\textbf{and } \phi_{1,n-1} \Rightarrow \bigvee\{\phi_{2,n-1} \mid (q_{2,n-1}, \emptyset, \phi_{2,n-1}, q_{2,n}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,n} \; R \; q_{2,n}\}\\
&\textbf{and } \phi_{1,n} \Rightarrow \bigvee\{\phi_{2,n} \mid (q_{2,n}, P_{1\star}, \phi_{2,n}, q_{2,n+1}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,n+1} \; R \; q_{2,n+1}\}\\
&\textbf{and } q_2 = q_{2,1}
\end{aligned}$$

Then, by applying Definition 26, conclude:

$$\begin{aligned}
&\mathsf{substNext}_1(\phi_{1,1}) \Rightarrow \bigvee\{\mathsf{substNext}_1(\phi_{2,1}) \mid (q_{2,1}, \emptyset, \phi_{2,1}, q_{2,2}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,2} \; R \; q_{2,2}\}\\
&\textbf{and } \mathsf{subst}_2(\phi_{1,2}) \Rightarrow \bigvee\{\mathsf{subst}_2(\phi_{2,2}) \mid (q_{2,2}, \emptyset, \phi_{2,2}, q_{2,3}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,3} \; R \; q_{2,3}\}\\
&\textbf{and } \cdots\\
&\textbf{and } \mathsf{subst}_{n-1}(\phi_{1,n-1}) \Rightarrow \bigvee\{\mathsf{subst}_{n-1}(\phi_{2,n-1}) \mid (q_{2,n-1}, \emptyset, \phi_{2,n-1}, q_{2,n}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,n} \; R \; q_{2,n}\}\\
&\textbf{and } \mathsf{substPrev}_n(\phi_{1,n}) \Rightarrow\\
&\quad \bigvee\{\mathsf{substPrev}_n(\phi_{2,n}) \mid (q_{2,n}, \emptyset, \phi_{2,n}, q_{2,n+1}) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q_{1,n+1} \; R \; q_{2,n+1}\}
\end{aligned}$$

Then, conclude:

$$\begin{aligned}
&\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n})\\
&\quad \Rightarrow \bigvee \left\{ \begin{array}{l|l} \begin{array}{l} \mathsf{substNext}_1(\phi_{2,1}) \\ \;\wedge \mathsf{subst}_2(\phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{2,n-1}) \\ \;\wedge \mathsf{substPrev}_n(\phi_{2,n}) \end{array} & \begin{array}{l} (q_{2,1}, \emptyset, \phi_{2,1}, q_{2,2}), \\ \ldots, \\ (q_{2,n-1}, \emptyset, \phi_{2,n-1}, q_{2,n}), \\ (q_{2,n}, \emptyset, \phi_{2,n}, q_{2,n+1}) \in \mathsf{Trans}(\alpha_2) \\ \textbf{and } q_{1,n+1} \; R \; q_{2,n+1} \end{array} \end{array} \right\}
\end{aligned}$$

Then, conclude:

$$\begin{aligned}
&\exists \mathsf{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))\\
&\quad \Rightarrow \bigvee \left\{ \begin{array}{l|l} \begin{array}{l} \exists \mathsf{Img}(\dagger).(\\ \quad \mathsf{substNext}_1(\phi_{2,1}) \\ \quad \wedge \mathsf{subst}_2(\phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{2,n-1}) \\ \quad \wedge \mathsf{substPrev}_n(\phi_{2,n}) \\ ) \end{array} & \begin{array}{l} (q_{2,1}, \emptyset, \phi_{2,1}, q_{2,2}), \\ \ldots, \\ (q_{2,n-1}, \emptyset, \phi_{2,n-1}, q_{2,n}), \\ (q_{2,n}, P_{1\star}, \phi_{2,n}, q_{2,n+1}) \\ \quad \in \mathsf{Trans}(\alpha_2) \\ \textbf{and } q_{1,n+1} \; R \; q_{2,n+1} \end{array} \end{array} \right\}
\end{aligned}$$

Then, because $P_{1\star} \neq \emptyset$, and by applying Definition 27, conclude:

$$\begin{aligned}
&\exists \mathsf{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))\\
&\quad \Rightarrow \bigvee\{\phi_{2\star} \mid (q_{2,1}, P_{1\star}, \phi_{2\star}, q_{2,n+1}) \in \mathsf{Trans}(\alpha_2^\star) \textbf{ and } q_{1,n+1} \; R \; q_{2,n+1}\}
\end{aligned}$$

Then, because $q_1 = q_{1,1}$, and because $q_1' = q_{1,n+1}$, and because $q_2 = q_{2,1}$, and because $\phi_{1\star} = \exists \mathsf{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))$, conclude:

$$\phi_{1\star} \Rightarrow \bigvee\{\phi_{2\star} \mid (q_2, P_{1\star}, \phi_{2\star}, q_{2,n+1}) \in \mathsf{Trans}(\alpha_2^\star) \textbf{ and } q_1' \; R \; q_{2,n+1}\}.$$

Then, by applying Definition 18, conclude $\alpha_1^\star \preceq_R \alpha_2^\star$.

- Case $\left[(q_1, P_{1\star}, \phi_{1\star}, q'_1) \in \mathsf{Trans}(\alpha_1) \textbf{ and } P_{1\star} \neq \emptyset\right]$. Then, because $\alpha_1 \simeq \alpha_2$, and because $q_1 \; R \; q_2$, and by applying Definitions 18, 20, and 21, and by applying induction, conclude:

  $$\phi_{1\star} \Rightarrow \bigvee \{\phi_2 \mid (q_2, P_{1\star}, \phi_2, q'_2) \in \mathsf{Trans}(\alpha_2) \textbf{ and } q'_1 \; R \; q'_2\}$$

  Then, because $P_{1\star} \neq \emptyset$, and by applying Definition 27, conclude:

  $$\phi_{1\star} \Rightarrow \bigvee \{\phi_2 \mid (q_2, P_{1\star}, \phi_2, q'_2) \in \mathsf{Trans}(\alpha_2^\star) \textbf{ and } q'_1 \; R \; q'_2\}$$

  Then, by applying Definition 18, conclude $\alpha_1^\star \preceq_R \alpha_2^\star$.

Thus, conclude $\alpha_1^\star \preceq_R \alpha_2^\star$. Symmetrically, conclude $\alpha_2^\star \preceq_{R^{-1}} \alpha_1^\star$. Then, by applying Definitions 20 and 21, conclude $\alpha_1^\star \simeq \alpha_2^\star$. □

Proof of Theorem 11. For some α, suppose $w \in \mathsf{Behav}(\alpha)$. Then, by applying Definition 16, for some $q, q', \ldots, w', w'', \ldots, \mu, \mu', \ldots$, conclude $\left[q \in \mathsf{Init}(\alpha) \textbf{ and } (q, w, \mu) \vdash_\alpha (q', w', \mu') \vdash_\alpha \cdots\right]$.

For some $n, \hat{q}_1, \ldots, \hat{q}_{n+1}, \hat{\lambda}, \hat{w}, \hat{\mu}_1, \ldots, \hat{\mu}_{n+1}$, suppose:

$$(\hat{q}_1, \hat{\lambda}\hat{w}, \hat{\mu}_1) \vdash_\alpha \cdots \vdash_\alpha (\hat{q}_n, \hat{\lambda}\hat{w}, \hat{\mu}_n) \vdash_\alpha (\hat{q}_{n+1}, \hat{w}, \hat{\mu}_{n+1})$$

Then, by applying Definition 15, for some $\hat{\phi}_1, \ldots, \hat{\phi}_n, \hat{P}$, conclude:

$$\begin{aligned}
&(\hat{q}_1, \emptyset, \hat{\phi}_1, \hat{q}_2), \ldots, (\hat{q}_{n-1}, \emptyset, \hat{\phi}_{n-1}, \hat{q}_n), (\hat{q}_n, \hat{P}, \hat{\phi}_n, \hat{q}_{n+1}) \in \mathsf{Trans}(\alpha)\\
&\textbf{and } \mathsf{Dom}(\hat{\mu}_1) = \cdots = \mathsf{Dom}(\hat{\mu}_{n-1}) = \mathsf{Memor}(\alpha) \textbf{ and } \mathsf{Dom}(\hat{\lambda}) = \hat{P} \neq \emptyset\\
&\textbf{and } \{{}^\bullet m \mapsto \hat{\mu}_1(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \hat{\phi}_1\\
&\textbf{and } \{{}^\bullet m \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_3(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \hat{\phi}_2\\
&\textbf{and } \cdots\\
&\textbf{and } \{{}^\bullet m \mapsto \hat{\mu}_{n-1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \hat{\phi}_{n-1}\\
&\textbf{and } \hat{\lambda} \cup \{{}^\bullet m \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_{n+1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \hat{\phi}_n
\end{aligned}$$

Then, conclude:

$$\begin{aligned}
&\{{}^\bullet m \mapsto \hat{\mu}_1(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{\dagger_1(m) \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \hat{\phi}_1\{[\dagger_1(m)/m^\bullet] \mid m \in \mathbb{M}\}\\
&\textbf{and } \{\dagger_1(m) \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{\dagger_2(m) \mapsto \hat{\mu}_3(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \models \hat{\phi}_2\{[\dagger_1(m)/{}^\bullet m] \mid m \in \mathbb{M}\}\{[\dagger_2(m)/m^\bullet] \mid m \in \mathbb{M}\}\\
&\textbf{and } \cdots\\
&\textbf{and } \{\dagger_{n-2}(m) \mapsto \hat{\mu}_{n-1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{\dagger_{n-1}(m) \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \models \hat{\phi}_{n-1}\{[\dagger_{n-2}(m)/{}^\bullet m] \mid m \in \mathbb{M}\}\{[\dagger_{n-1}(m)/m^\bullet] \mid m \in \mathbb{M}\}\\
&\textbf{and } \hat{\lambda} \cup \{\dagger_{n-1}(m) \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_{n+1}(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \models \hat{\phi}_n\{[\dagger_{n-1}/{}^\bullet m] \mid m \in \mathbb{M}\}
\end{aligned}$$

Then, by applying Definition 26, conclude:

$$\begin{aligned}
&\{{}^\bullet m \mapsto \hat{\mu}_1(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{\dagger_1(m) \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \mathsf{substNext}_1(\hat{\phi}_1)\\
&\textbf{and } \{\dagger_1(m) \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{\dagger_2(m) \mapsto \hat{\mu}_3(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \mathsf{subst}_2(\hat{\phi}_2)\\
&\textbf{and } \cdots\\
&\textbf{and } \{\dagger_{n-2}(m) \mapsto \hat{\mu}_{n-1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{\dagger_{n-1}(m) \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \mathsf{subst}_{n-1}(\hat{\phi}_{n-1})\\
&\textbf{and } \hat{\lambda} \cup \{\dagger_{n-1}(m) \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_{n+1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \mathsf{substPrev}_n(\hat{\phi}_n)
\end{aligned}$$

Then, conclude:

$$\begin{aligned}
&\hat{\lambda} \cup \{{}^\bullet m \mapsto \hat{\mu}_1(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \cup \{\dagger_1(m) \mapsto \hat{\mu}_2(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \cup \cdots\\
&\quad \cup \{\dagger_{n-1}(m) \mapsto \hat{\mu}_n(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \cup \{m^\bullet \mapsto \hat{\mu}_{n+1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \mathsf{substNext}_1(\hat{\phi}_1)\\
&\qquad\qquad\qquad\qquad \wedge \mathsf{subst}_2(\hat{\phi}_2) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\hat{\phi}_{n-1})\\
&\qquad\qquad\qquad\qquad \wedge \mathsf{substPrev}_n(\hat{\phi}_n)
\end{aligned}$$

Then, conclude:

$$\begin{aligned}
&\hat{\lambda} \cup \{{}^\bullet m \mapsto \hat{\mu}_1(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_{n+1}(m) \mid m \in \mathsf{Memor}(\alpha)\}\\
&\quad \models \exists \mathsf{Img}(\dagger).(\mathsf{substNext}_1(\hat{\phi}_1) \wedge \mathsf{subst}_2(\hat{\phi}_2) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\hat{\phi}_{n-1}) \wedge \mathsf{substPrev}_n(\hat{\phi}_n))
\end{aligned}$$

Because $(\hat{q}_1, \emptyset, \hat{\phi}_1, \hat{q}_2), \ldots, (\hat{q}_{n-1}, \emptyset, \hat{\phi}_{n-1}, \hat{q}_n), (\hat{q}_n, \hat{P}, \hat{\phi}_n, \hat{q}_{n+1}) \in \mathsf{Trans}(\alpha)$, and because $\hat{P} \neq \emptyset$, and because $\mathsf{Dom}(\hat{\mu}_1) = \mathsf{Dom}(\hat{\mu}_{n+1}) = \mathsf{Memor}(\alpha)$, and by applying Definition 27, conclude:

$$\begin{array}{l}(\hat{q}_1, \hat{P}, \exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\hat{\phi}_1) \wedge \mathsf{subst}_2(\hat{\phi}_2) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\hat{\phi}_{n-1}) \wedge \mathsf{substPrev}_n(\hat{\phi}_n)), \hat{q}_{n+1}) \in \mathsf{Trans}(\alpha^\star) \\ \textbf{and } \mathsf{Dom}(\hat{\mu}_1) = \mathsf{Dom}(\hat{\mu}_{n+1}) = \mathsf{Memor}(\alpha^\star) \textbf{ and } \mathsf{Init}(\alpha) = \mathsf{Init}(\alpha^\star)\end{array}$$

Then, because $\hat{\lambda} \cup \{{}^\bullet m \mapsto \hat{\mu}_1(m) \mid m \in \mathsf{Memor}(\alpha)\} \cup \{m^\bullet \mapsto \hat{\mu}_{n+1}(m) \mid m \in \mathsf{Memor}(\alpha)\} \models \exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\hat{\phi}_1) \wedge \mathsf{subst}_2(\hat{\phi}_2) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\hat{\phi}_{n-1}) \wedge \mathsf{substPrev}_n(\hat{\phi}_n))$, and by Definition 15, conclude $(\hat{q}_1, \hat{\lambda}\hat{w}, \hat{\mu}_1) \vdash_{\alpha^\star} (\hat{q}_{n+1}, \hat{w}, \hat{\mu}_{n+1})$.

Thus, for all $n, \hat{q}_1, \ldots, \hat{q}_{n+1}, \hat{\lambda}, \hat{w}, \hat{\mu}_1, \ldots, \hat{\mu}_{n+1}$, conclude:

$$\begin{array}{l}(\hat{q}_1, \hat{\lambda}\hat{w}, \hat{\mu}_1) \vdash_\alpha \cdots \vdash_\alpha (\hat{q}_n, \hat{\lambda}\hat{w}, \hat{\mu}_n) \vdash_\alpha (\hat{q}_{n+1}, \hat{w}, \hat{\mu}_{n+1}) \\ \quad \textbf{implies } (\hat{q}_1, \hat{\lambda}\hat{w}, \hat{\mu}_1) \vdash_{\alpha^\star} (\hat{q}_{n+1}, \hat{w}, \hat{\mu}_{n+1})\end{array}$$

Then, because $(q, w, \mu) \vdash_\alpha \cdots$, and by applying induction, conclude $(q, w, \mu) \vdash_{\alpha^\star} \cdots$. Then, because $q \in \mathsf{Init}(\alpha)$, and because $\mathsf{Init}(\alpha) = \mathsf{Init}(\alpha^\star)$, and by applying Definition 16, conclude $w \in \mathsf{Behav}(\alpha^\star)$. Then, conclude $\mathsf{Behav}(\alpha) \subseteq \mathsf{Behav}(\alpha^\star)$. Analogously, conclude $\mathsf{Behav}(\alpha^\star) \subseteq \mathsf{Behav}(\alpha)$. Then, by applying Definition 17, conclude $\alpha \approx \alpha^\star$. □

Proof of Theorem 12. For some α_1, α_2, suppose $\big[\mathsf{IdlingEnabled}(\alpha_1) \textbf{ and } \mathsf{IdlingEnabled}(\alpha_2)\big]$.

Let $R = \{((q_1, q_2), (q_1, q_2)) \mid q_1 \in \mathsf{Stat}(\alpha_1) \textbf{ and } q_2 \in \mathsf{Stat}(\alpha_2)\}$.

For some $q_{12\star}, q'_{12\star} q_{1\star2\star}, P_{12\star}, \phi_{12\star}$, suppose:

$$\big[(q_{12\star}, P_{12\star}, \phi_{12\star}, q'_{12\star}) \in \mathsf{Trans}((\alpha_1 \otimes \alpha_2)^\star) \textbf{ and } q_{12\star} \; R \; q_{1\star2\star}\big]$$

Then, by applying Definitions 22 and 27, for some $n, q_{1,1}, \ldots, q_{1,n}, q_{2,1}, \ldots, q_{2,n}, \phi_{1,1}, \ldots, \phi_{1,n}, \phi_{2,1}, \ldots, \phi_{2,n}, P_1, P_2$, conclude:

$$\begin{array}{l}(q_{1,1}, \emptyset, \phi_{1,1}, q_{1,2}), \ldots, (q_{1,n-1}, \emptyset, \phi_{1,n-1}, q_{1,n}), (q_{1,n}, P_1, \phi_{1,n}, q_{1,n+1}) \in \mathsf{Trans}(\alpha_1) \\ \textbf{and } (q_{2,1}, \emptyset, \phi_{2,1}, q_{2,2}), \ldots, (q_{2,n-1}, \emptyset, \phi_{2,n-1}, q_{2,n}), (q_{1,n}, P_2, \phi_{2,n}, q_{2,n+1}) \in \mathsf{Trans}(\alpha_2) \\ \textbf{and } P_1 \neq \emptyset \textbf{ and } P_2 \neq \emptyset \textbf{ and } \mathsf{Port}(\alpha_1) \cap P_2 = \mathsf{Port}(\alpha_2) \cap P_1 \\ \textbf{and } q_{12\star} = (q_{1,1}, q_{2,1}) \textbf{ and } q'_{12\star} = (q_{1,n}, q_{2,n}) \textbf{ and } P_{12\star} = P_1 \cup P_2 \\ \textbf{and } \phi_{12\star} = \exists \mathrm{Img}(\dagger).(\\ \qquad\qquad \mathsf{substNext}_1(\phi_{1,1} \wedge \phi_{2,1}) \\ \qquad\qquad \wedge\, \mathsf{subst}_2(\phi_{1,2} \wedge \phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1} \wedge \phi_{2,n-1}) \\ \qquad\qquad \wedge\, \mathsf{substPrev}_n(\phi_{1,n} \wedge \phi_{2,n}) \\ \qquad\quad)\end{array}$$

Then, by applying Definitions 22 and 27, conclude:

$$\left(\begin{array}{c}(q_{1,1}, q_{2,1}), \\ P_{12\star}, \\ (\exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))) \\ \wedge\, (\exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\phi_{2,1}) \wedge \mathsf{subst}_2(\phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{2,n-1}) \wedge \mathsf{substPrev}_n(\phi_{2,n}))), \\ (q_{1,n+1}, q_{2,n+1})\end{array}\right)$$
$$\in \mathsf{Trans}(\alpha_1^\star \otimes \alpha_2^\star)$$

Then, because $\phi_{12\star} = \exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1} \wedge \phi_{2,1}) \wedge \mathsf{subst}_2(\phi_{1,2} \wedge \phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1} \wedge \phi_{2,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n} \wedge \phi_{2,n}))$, conclude $\phi_{12\star} \Rightarrow \bigvee\{\phi_{1\star2\star} \mid ((q_{1,1}, q_{2,1}), P_{12\star}, \phi_{1\star2\star}, (q_{1,n+1}, q_{2,n+1})) \in \mathsf{Trans}(\alpha_1^\star \otimes \alpha_2^\star)\}$. Then, because $q_{12\star} \; R \; q_{1\star2\star}$, and because $q_{12\star} = (q_{1,1}, q_{2,1})$, and because $q'_{12\star} = (q_{1,n+1}, q_{2,n+1})$, and by the definition of R, conclude $\phi_{12\star} \Rightarrow \bigvee\{\phi_{1\star2\star} \mid (q_{1\star2\star}, P_{12\star}, \phi_{1\star2\star}, (q_{1,n+1}, q_{2,n+1})) \in \mathsf{Trans}(\alpha_1^\star \otimes \alpha_2^\star) \textbf{ and } q'_{12\star} \; R \; (q_{1,n+1}, q_{2,n+1})\}$. Then, by applying Definition 18, conclude $(\alpha_1 \otimes \alpha_2)^\star \preceq_R \alpha_1^\star \otimes \alpha_2^\star$.

The proof that $\alpha_1^\star \otimes \alpha_2^\star \preceq_{R^{-1}} (\alpha_1 \otimes \alpha_2)^\star$ is analogous except for two complications. First, handle:

$$\begin{array}{l}(q_{1,1}, \emptyset, \phi_{1,1}, q_{1,2}), \ldots, (q_{1,n-1}, \emptyset, \phi_{1,n_1-1}, q_{1,n_1}), (q_{1,n_1}, P_1, \phi_{1,n_1}, q_{1,n_1+1}) \in \mathsf{Trans}(\alpha_1) \\ \textbf{and } (q_{2,1}, \emptyset, \phi_{2,1}, q_{2,2}), \ldots, (q_{2,n_2-1}, \emptyset, \phi_{2,n_2-1}, q_{2,n_2}), (q_{1,n_2}, P_2, \phi_{2,n_2}, q_{2,n_2+1}) \in \mathsf{Trans}(\alpha_2) \\ \textbf{and } n_1 \neq n_2\end{array}$$

by using $\mathsf{IdlingEnabled}(\alpha_1)$ or $\mathsf{IdlingEnabled}(\alpha_2)$ to prefix the shortest chain with idling transitions until both chains are of equal length. Second, conclude:

$$\begin{array}{l}(\exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))) \\ \wedge\, (\exists \mathrm{Img}(\dagger).(\mathsf{substNext}_1(\phi_{2,1}) \wedge \mathsf{subst}_2(\phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{2,n-1}) \wedge \mathsf{substPrev}_n(\phi_{2,n}))) \\ \quad \Rightarrow \exists \mathrm{Img}(\dagger).(\\ \qquad\quad \mathsf{substNext}_1(\phi_{1,1} \wedge \phi_{2,1}) \\ \qquad\quad \wedge\, \mathsf{subst}_2(\phi_{1,2} \wedge \phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1} \wedge \phi_{2,n-1}) \\ \qquad\quad \wedge\, \mathsf{substPrev}_n(\phi_{1,n} \wedge \phi_{2,n}) \\ \qquad)\end{array}$$

by observing that $\mathrm{Img}(\dagger) \cap \mathsf{Free}(\mathsf{substNext}_1(\phi_{1,1}) \wedge \mathsf{subst}_2(\phi_{1,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{1,n-1}) \wedge \mathsf{substPrev}_n(\phi_{1,n}))$ and $\mathrm{Img}(\dagger) \cap \mathsf{Free}(\mathsf{substNext}_1(\phi_{2,1}) \wedge \mathsf{subst}_2(\phi_{2,2}) \wedge \cdots \wedge \mathsf{subst}_{n-1}(\phi_{2,n-1}) \wedge \mathsf{substPrev}_n(\phi_{2,n}))$ are disjoint (because $\mathsf{Memor}(\alpha_1)$ and $\mathsf{Memor}(\alpha_2)$ are disjoint).

Thus, conclude $(\alpha_1 \otimes \alpha_2)^\star \preceq_R \alpha_1^\star \otimes \alpha_2^\star$ and $\alpha_1^\star \otimes \alpha_2^\star \preceq_{R^{-1}} (\alpha_1 \otimes \alpha_2)^\star$. Then, by applying Definitions 20 and 21, conclude $(\alpha_1 \otimes \alpha_2)^\star \simeq \alpha_1^\star \otimes \alpha_2^\star$. □

Proof of Theorem 13. For a detailed proof, we refer to our technical report [35, Theorem 5, p. 33]. □

Appendix C. Proofs for Section 6

Proof of Theorem 14. By applying Definition 30, and by applying Theorem 3, conclude that $\approxeq$ is reflexive, symmetric, and transitive. □

Proof of Theorem 15. For some α_1, α_2, suppose $\alpha_1 \simeq \alpha_2$. Then, by applying Theorem 10, conclude $\alpha_1^\star \simeq \alpha_2^\star$. Then, by applying Definition 30, conclude $\alpha_1 \approxeq \alpha_2$. □

Proof of Theorem 16. For some α_1, α_2, suppose $\alpha_1 \approxeq \alpha_2$. Then, by applying Definition 30, conclude $\alpha_1^\star \simeq \alpha_2^\star$. Then, by applying Theorem 4, conclude $\alpha_1^\star \approx \alpha_2^\star$. Then, by applying Theorem 11, conclude $\alpha_1 \approx \alpha_1^\star \approx \alpha_2^\star \approx \alpha_2$. Then, by applying Theorem 1, conclude $\alpha_1 \approx \alpha_2$. □

Proof of Theorem 17. For some $\alpha_1, \alpha_2, \alpha_3, \alpha_4$, suppose:

$$\alpha_1 \approxeq \alpha_2 \textbf{ and } \alpha_3 \approxeq \alpha_4 \textbf{ and } \mathsf{IdlingEnabled}(\alpha_1 \otimes \alpha_3) \textbf{ and } \mathsf{IdlingEnabled}(\alpha_2 \otimes \alpha_4)$$

Then, by applying Definition 30, conclude $\left[\alpha_1^\star \simeq \alpha_2^\star \textbf{ and } \alpha_3^\star \simeq \alpha_4^\star\right]$. Then, by applying Theorem 7, conclude $\alpha_1^\star \otimes \alpha_3^\star \simeq \alpha_2^\star \otimes \alpha_4^\star$. Then, because $\mathsf{IdlingEnabled}(\alpha_1 \otimes \alpha_3)$, and because $\mathsf{IdlingEnabled}(\alpha_2 \otimes \alpha_4)$, and by applying Theorem 12, conclude $(\alpha_1 \otimes \alpha_3)^\star \simeq (\alpha_2 \otimes \alpha_4)^\star$. Then, by applying Definition 30, conclude:

$$\alpha_1 \otimes \alpha_3 \approxeq \alpha_2 \otimes \alpha_4 \qquad \square$$

Proof of Theorem 18. For some α, let $R = \{(q, q) \mid q \in \mathsf{Stat}(\alpha)\}$. Then, by applying Definition 27, conclude $\alpha^\star \simeq_R \alpha^{\star\star}$. Then, by applying Definition 21, conclude $\alpha^\star \simeq \alpha^{\star\star}$. Then, by applying Definition 30, conclude $\alpha \approxeq \alpha^\star$. □

Proof of Theorem 19. By its definition, P_{12} contains the internal ports of $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$. Consequently, no port in P_{12} is shared with any other constraint automaton. Then, by applying Definitions 23 and 24, conclude $((\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \ominus P_{12}) \otimes \boldsymbol{a}_3 = ((\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \ominus P_{12}) \otimes (\boldsymbol{a}_3 \ominus P_{12})$. Then, by applying Theorem 9, conclude:

$$((\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \ominus P_{12}) \otimes \boldsymbol{a}_3 \simeq ((\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \otimes \boldsymbol{a}_3) \ominus P_{12}$$

In this way, "move" all port subtractions outward to conclude:

$$((\cdots(((\alpha_1 \otimes \alpha_2) \ominus P_{12}) \cdots) \otimes \alpha_n) \ominus P_{(12)\cdots n}) \simeq ((\alpha_1 \otimes \cdots \otimes \alpha_n) \ominus P_{12} \ominus \cdots \ominus P_{(12)\cdots n})$$

Then, by applying Theorem 15, conclude:

$$((\cdots(((\alpha_1 \otimes \alpha_2) \ominus P_{12}) \cdots) \otimes \alpha_n) \ominus P_{(12)\cdots n}) \approxeq ((\alpha_1 \otimes \cdots \otimes \alpha_n) \ominus P_{12} \ominus \cdots \ominus P_{(12)\cdots n})$$

Then, by applying Theorems 8, 17 and 18, conclude:

$$((\cdots(((\alpha_1 \otimes \alpha_2) \ominus P_{12})^\star \cdots) \otimes \alpha_n) \ominus P_{(12)\cdots n})^\star \approxeq ((\alpha_1 \otimes \cdots \otimes \alpha_n) \ominus P_{12} \ominus \cdots \ominus P_{(12)\cdots n})^\star$$

Finally, by applying Theorems 8, 13, 15 and 17, conclude:

$$\lfloor((\cdots(\lfloor((\alpha_1 \otimes \alpha_2) \ominus P_{12})^\star\rfloor \cdots) \otimes \alpha_n) \ominus P_{(12)\cdots n})^\star\rfloor \approxeq \lfloor((\alpha_1 \otimes \cdots \otimes \alpha_n) \ominus P_{12} \ominus \cdots \ominus P_{(12)\cdots n})^\star\rfloor \qquad \square$$

References

[1] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.
[2] F. Arbab Puff, The Magic Protocol, in: G. Agha, O. Danvy, J. Meseguer (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems (Talcott Festschrift), in: LNCS, vol. 7000, Springer, 2011, pp. 169–206.
[3] S.-S. Jongmans, F. Arbab, Overview of thirty semantic formalisms for Reo, Sci. Ann. Comput. Sci. 22 (1) (2012) 201–251.
[4] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in Reo by constraint automata, Sci. Comput. Program. 61 (2) (2006) 75–113.
[5] S.-S. Jongmans, Automata-Theoretic Protocol Programming, Ph.D. thesis, Leiden University, 2016.
[6] 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.
[7] S.-S. Jongmans, F. Santini, F. Arbab, Partially-distributed coordination with Reo and constraint automata, Serv. Oriented Comput. Appl. 9 (3) (2015) 311–339.

[8] S.-S. Jongmans, F. Arbab, Can high throughput atone for high latency in compiler-generated protocol code?, in: M. Dastani, M. Sirjani (Eds.), Fundamentals of Software Engineering (Proceedings of FSEN 2015), in: LNCS, vol. 9392, Springer, 2015, pp. 238–258.
[9] C. Krause, H. Giese, E. de Vink, Compositional and behavior-preserving reconfiguration of component connectors in Reo, J. Vis. Lang. Comput. 24 (3) (2013) 153–168.
[10] B. Pourvatan, M. Sirjani, F. Arbab, M. Bonsangue, Decomposition of constraint automata, in: L. Barbosa, M. Lumpe (Eds.), Formal Aspects of Component Software (Proceedings of FACS 2010), in: LNCS, vol. 6921, Springer, 2012, pp. 237–258.
[11] B. Pourvatan, M. Sirjani, H. Hojjat, F. Arbab, Symbolic execution of Reo circuits using constraint automata, Sci. Comput. Program. 77 (7–8) (2012) 848–869.
[12] S.-S. Jongmans, T. Kappé, F. Arbab, Composing constraint automata, state-by-state, in: C. Braga, P. Ölveczky (Eds.), Formal Aspects of Component Software (Proceedings of FACS 2015), in: LNCS, vol. 9539, Springer, 2016, pp. 217–236.
[13] W. Reisig, Introductory examples and basic definitions, in: Petri Nets: An Introduction, in: EATCS Monographs on Theoretical Computer Science, vol. 4, Springer, 1985, pp. 3–16, Ch. 1.
[14] W. Rautenberg, First-order logic, in: A Concise Introduction to Mathematical Logic, 3rd edition, in: Universitext, Springer, 2010, pp. 41–90, Ch. 2.
[15] F. Arbab, F. Santini, Preference and similarity-based behavioral discovery of services, in: Web Services and Formal Methods (Proceedings of WS-FM 2012), in: LNCS, vol. 7843, Springer, 2013, pp. 118–133.
[16] T. Kappé, F. Arbab, C. Talcott, A compositional framework for preference-aware agents, in: Proceedings of V2CPS 2016, 2016.
[17] S. Chatterjee, M. Kishinevsky, U. Ogras, xMAS: quick formal modeling of communication fabrics to enable verification, IEEE Des. Test Comput. 29 (3) (2012) 80–88, http://dx.doi.org/10.1109/MDT.2012.2205998.
[18] C. Koehler, D. Clarke, Decomposing port automata, in: M. Schumacher, A. Wood (Eds.), Proceedings of SAC 2009, ACM, 2009, pp. 1369–1373.
[19] S. Bliudze, J. Sifakis, The algebra of connectors—structuring interaction in BIP, IEEE Trans. Comput. 57 (10) (2008) 1315–1330.
[20] S. Bliudze, J. Sifakis, Causal semantics for the algebra of connectors, Form. Methods Syst. Des. 36 (2) (2010) 167–194.
[21] M. Izadi, M. Bonsangue, D. Clarke, Büchi automata for modeling component connectors, Softw. Syst. Model. 10 (2) (2011) 183–200.
[22] M. Izadi, Model Checking of Component Connectors, Ph.D. thesis, Leiden University, 2011.
[23] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, W. Leister, Design and verification of systems with exogenous coordination using vereofy, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification, and Validation (Proceedings of ISoLA 2010), in: LNCS, vol. 6416, Springer, 2010, pp. 97–111.
[24] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, A uniform framework for modeling and verifying components and connectors, in: J. Field, V. Vasconcelos (Eds.), Coordination Models and Languages (Proceedings of COORDINATION 2009), in: LNCS, vol. 5521, Springer, 2009, pp. 247–267.
[25] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, Formal verification for components and connectors, in: F. de Boer, M. Bonsangue, E. Madelaine (Eds.), Formal Methods for Components and Objects (Proceedings of FMCO 2008), in: LNCS, vol. 5751, Springer, 2009, pp. 82–101.
[26] C. Baier, J. Klein, S. Klüppelholz, Modeling and verification of components and connectors, in: M. Bernardo, V. Issarny (Eds.), Formal Methods for Eternal Networked Software Systems (Proceedings of SFM 2011), in: LNCS, vol. 6659, Springer, 2011, pp. 114–147.
[27] J. Klein, Compositional Synthesis and Most General Controllers, Ph.D. thesis, Dresden University of Technology, 2012.
[28] S. Klüppelholz, C. Baier, Symbolic model checking for channel-based component connectors, Sci. Comput. Program. 74 (9) (2009) 688–701.
[29] S. Klüppelholz, C. Baier, Alternating-time stream logic for multi-agent systems, Sci. Comput. Program. 75 (6) (2010) 398–425.
[30] S. Klüppelholz, Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models, Ph.D. thesis, Dresden University of Technology, 2012.
[31] 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.
[32] F. Arbab, J. Rutten, A coinductive calculus of component connectors, in: M. Wirsing, D. Pattinson, R. Hennicker (Eds.), Recent Trends in Algebraic Development Techniques (Proceedings of WADT 2002), in: LNCS, vol. 2755, Springer, 2003, pp. 34–55.
[33] F. Arbab, Abstract behavior types: a foundation model for components and their composition, Sci. Comput. Program. 55 (1–3) (2005) 3–52.
[34] D. Clarke, D. Costa, F. Arbab, Connector colouring I: synchronisation and context dependency, Sci. Comput. Program. 66 (3) (2007) 205–225.
[35] S.-S. Jongmans, T. Kappé, F. Arbab, Composing Constraint Automata, State-by-State (Technical Report), Tech. Rep. FM-1506, CWI, 2015, http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-23621.
[36] S.-S. Jongmans, F. Arbab, Toward sequentializing overparallelized protocol code, in: I. Lanese, A. Lluch-Lafuente, A. Sokolova, H.T. Vieira (Eds.), Proceedings of ICE 2014, in: EPTCS, vol. 166, 2014, pp. 38–44, CoRR.
[37] J. Proença, Synchronous Coordination of Distributed Components, Ph.D. thesis, Universiteit Leiden, 2011.
[38] F. Ghassemi, S. Tasharofi, M. Sirjani, Automated mapping of Reo circuits to constraint automata, in: F. Arbab, M. Sirjani (Eds.), Foundations of Software Engineering (Proceedings of FSEN 2005), in: ENTCS, vol. 159, Elsevier, 2006, pp. 99–115.
[39] B. Pourvatan, N. Rouhy, An alternative algorithm for constraint automata product, in: F. Arbab, M. Sirjani (Eds.), Fundamentals of Software Engineering (Proceedings of FSEN 2007), in: LNCS, vol. 4767, Springer, 2007, pp. 412–422.
[40] J. Hopcroft, R. Motwani, J. Ullman, Finite automata, in: Introduction to Automata Theory, Languages, and Computation, 2nd edition, Addison-Wesley, 2001, pp. 37–81, Ch. 2.
[41] R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: P. Dembinski, M. Sredniawa (Eds.), Protocol Specification, Testing and Verification XV (Proceedings of PSTV 1995), in: IFIP AICT, Springer, 1995, pp. 3–18.
[42] D. Costa, Formal Models for Component Connectors, Ph.D. thesis, Vrije Universiteit Amsterdam, 2010.
[43] M. Sirjani, M.-M. Jaghoori, C. Baier, F. Arbab, Compositional semantics of an actor-based language using constraint automata, in: P. Ciancarini, H. Wiklicky (Eds.), Coordination Models and Languages (Proceedings of COORDINATION 2006), in: LNCS, vol. 4038, Springer, 2006, pp. 281–297.
[44] M. Sirjani, A. Movaghar, A. Shali, F. de Boer, Modeling and verification of reactive systems using Rebeca, Fundam. Inform. 63 (4) (2004) 385–410.
[45] F. Arbab, N. Kokash, S. Meng, Towards using Reo for compliance-aware business process modeling, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation (Proceedings of ISoLA 2008), in: CCIS, vol. 17, Springer, 2008, pp. 108–123.
[46] B. Changizi, N. Kokash, F. Arbab, A unified toolset for business process model formalization, in: B. Buhnova, J. Happe (Eds.), Preproceedings of FESCA 2010, 2010, pp. 147–156.
[47] S. Meng, F. Arbab, C. Baier, Synthesis of Reo circuits from scenario-based interaction specifications, Sci. Comput. Program. 76 (8) (2011) 651–680.
[48] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: D.V. Hung, P. Pandya (Eds.), Proceedings of SEFM 2006, IEEE, 2006, pp. 3–12.
[49] K. Dokter, S.-S. Jongmans, F. Arbab, S. Bliudze, Relating BIP and Reo, in: S. Knight, I. Lanese, A. Lluch-Lafuente, H.T. Vieira (Eds.), ICE 2015, in: EPTCS, vol. 189, 2015, pp. 3–20, CoRR.
[50] K. Dokter, S.-S. Jongmans, F. Arbab, S. Bliudze, Combine and conquer: relating BIP and Reo, J. Log. Algebraic Methods Program. 86 (2017) 134–156, http://dx.doi.org/10.1016/j.jlamp.2016.09.008.