Handshaking Protocol for Distributed Implementation of Reo

N. Kokash
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
natallia.kokash@gmail.com

Reo, an exogenous channel-based coordination language, is a model for service coordination wherein services communicate through connectors formed by joining binary communication channels. In order to establish transactional communication among services as prescribed by connector semantics, distributed ports exchange "handshaking" messages signalling which parties are ready to provide or consume data. In this paper, we present a formal implementation model for distributed Reo with communication delays and outline ideas for its proof of correctness. To reason about Reo implementation formally, we introduce Timed Action Constraint Automata (TACA) and explain how to compare TACA with existing automata-based semantics for Reo. We use TACA to describe "handshaking" behavior of Reo modeling primitives and argue that in any distributed circuit remote Reo nodes and channels exposing such behavior commit to perform transitions envisaged by the network semantics.

1 Introduction

Service-oriented systems (SOS) are composed of autonomous services deployed on remote machines and accessed through the network. Reo coordination language [1] is an extensible notation for compositional modeling and execution of SOS. Services that have no prior knowledge about each other communicate through channel connectors which guarantee that each participant, service or client, receives right data at the right time. Each channel is a binary function that imposes synchronization and data constraints on input and output messages. Channels can be composed to realize complex behavioral protocols, including multi-party synchronous rendezvous. This approach enables models that are both concise and compositional, but it also makes operational semantics for Reo non-trivial.

The most basic semantic model for Reo is constraint automata (CA) [6]. States or locations in CA represent configurations of data stored in the buffers of Reo networks, while transition labels are composed of (i) sets of channel ends where dataflow is observed simultaneously, and (ii) data constraints necessary to trigger such transitions. The CA for a Reo connector can be computed as a product of the CA for its parts (sub-connectors or channels). CA is the theoretical basis for validation and verification tools for Reo, which are integrated in a framework known as the Extensible Coordination Tools (ECT)[1].

The Quality of Service (QoS) of a SOS depends on the quality of its components, efficiency of the "glue code" that coordinates individual services, and quality of the communication network. To evaluate the QoS of SOS coordinated by Reo, we need to estimate time to deliver input messages supplied by services to the input ports of the circuit to their consumers - services listening to the output ports of the circuit. The early semantic models for quantitative Reo [4, 3] assumed that delays in channels do not affect the operational semantics of Reo. This assumption is not realistic and limits the degree of concurrency in the presence of transactions with different durations. Hence, a more refined semantic model for Reo [16] was introduced to solve this problem.

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

Javier Cámara and José Proença (Eds.):
13th International Workshop on Foundations of
Coordination Languages and Self-Adaptive Systems (FOCLASA'14)
EPTCS 175, 2015, pp. 1–17, doi:10.4204/EPTCS.175.1

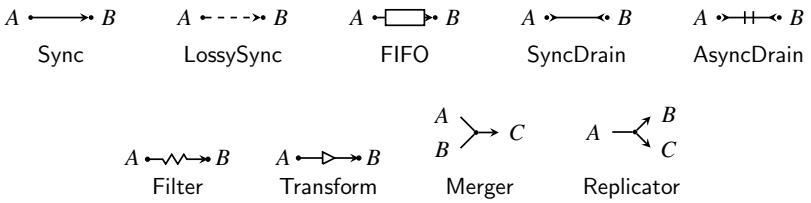


Figure 1: Graphical representation of basic Reo channels and nodes

Distributed Reo ports exchange technical messages signalling the readiness of coordinated services to provide or consume data. It has been recognized that the implementation of Reo should be distributed to avoid performance bottlenecks [21]. The existing semantic models for Reo focus on the description of the observable dataflow and are not suitable for the implementation and QoS evaluation. In this paper, we present a formal coordination protocol that can serve as foundational basis for distributed time-aware Reo implementation. One of the main issues is to decide which requests are considered simultaneous and should synchronize on each execution cycle. In our approach, we propose to use a timeout which each node in a network should wait for to acquire the information about pending requests on remote ports and decide which transition to fire. The timeout is chosen to guarantee that the information about the communication requests on boundary ports is propagated through the circuit.

The remainder of this paper is organized as follows. In Section 2, we explain the basics of Reo. In Section 3, we describe a semantic model for Reo used and extended in this paper. In Section 4, we explain the objectives of our work. In Section 5, we present implementation semantics for basic types of Reo nodes and channels. In Section 6, we explore the properties of our approach. Section 7 overviews related work. Finally, Section 8 concludes the paper and outlines future work.

2 Reo Coordination Language

Reo is a coordination language in which components and services are coordinated exogenously by channel-based connectors [1]. Connectors are graphs where the edges are user-defined communication channels and the nodes adhere to fixed routing rules. Channels in Reo are entities that have exactly two ends, also referred to as ports, which can be either *source* or *sink* ends. Source ends accept data into, and sink ends dispense data out of their channel.

Although channels can be defined by users, a set of basic Reo channels (see Figure 1) suffices to implement most common workflow patterns [5]. Among these channels are (i) the Sync channel, which is a directed channel that accepts a data item through its source end if it can instantly dispense it through its sink end; (ii) the LossySync channel, which always accepts a data item through its source end and tries to instantly dispense it through the sink end. If this is not possible, the data item is lost; (iii) the SyncDrain channel, which is a channel with two source ends that accept data simultaneously and loses them subsequently; (iv) the AsyncDrain channel, which accepts data items only through one of its two source channel ends at a moment in time and loses it; and (v) the FIFO channel, which is an asynchronous channel with a buffer of capacity one. For data manipulation, Reo introduces the Filter channel. which always accepts a data item at its source end and synchronously passes or loses it depending on whether or not the data item matches a certain predefined pattern or data constraint, and the Transform channel, which applies a user-defined function to the data item at its source end and synchronously yields the result at its sink end.

Channels can be joined together using nodes. A node can be a *source*, a *sink* or a *mixed* node. Source and sink nodes form the *boundary* nodes of a connector to enable the interaction with its environment.

Source nodes act as synchronous *replicators*, and sink nodes as non-deterministic *mergers*. A mixed node combines these two behaviors by atomically consuming a data item from one of its sink ends at the time and replicating it to all of its source ends. Often two other nodes, *route* and *join*, are used to model non-deterministic routing and synchronization of flow, respectively. The router can be constructed from basic Reo channels while the *join* node is a shorthand notation for a component that forms a tuple from data items received from several channel sink ports.

Channels can also differ at the level of their QoS. In quantitative Reo [4], channels are characterized by a set of associated QoS parameters such as communication delays or cost. We recognize two types of communication delays: *handshaking delay*, or time to decide whether the connector can satisfy the I/O request on its ends, and *data transfer delay*, or the time needed to transfer the data accepted by the circuit.

3 Semantic models for Reo

The semantics of any Reo connector can be better understood in terms of a specific semantic model and its appropriate translation into that model.

The most basic semantic model for Reo is constraint automata (CA) [6]. Transitions in CA are labeled with sets of ports that fire synchronously and data constraints on these ports. For example, a CA for the Sync channel with port ends A and B contains one state (s_0) and one transition $s_0 \xrightarrow{d_A=d_B,\{A,B\}} s_0$. The FIFO channel with ports A and B is described by a CA with two states corresponding to an empty buffer (s_0) and a full buffer (s_1), and transitions $s_0 \xrightarrow{d=d_A,\{A\}} s_1$ and $s_1 \xrightarrow{d_B=d,\{B\}} s_0$. The behavior of any Reo circuit can be computed using the product of CA of its basic channels. The *hiding operator* is introduced to abstract from unnecessary details such as dataflow on the internal ports [6].

Timed constrained automata (TCA) [2] represent CA with clock assignments and timing constraints.

The semantic models for Reo have been extended to compositionally compute QoS [4, 3], including communication delays in the circuit. It was assumed that delays do not affect operational semantics of the circuit and QoS labels were added to the transitions of basic CA. However, an example in [16] shows that such approach limits concurrency in the circuit. Action constraint automata (ACA) were introduced to overcome this issue [16]. ACA distinguish several kinds of actions triggered on channel ports to signal the state changes of the channel:

Definition 3.1 (ACA [16]) *An action constraint automaton $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ consists of a set of states S, a set of action names $\mathscr{N}$ derived from a set of port names $\mathscr{M}$ and a set of admissible action types $\mathscr{T}$, a transition relation $\rightarrow \subseteq S \times 2^{\mathscr{N}} \times DC \times S$, where DC is the set of data constraints over a finite data domain Data, and an initial state $s_0 \in S$.*

An ACA model proposed in [16] uses the set of action types $\mathscr{T}_1 = \{b, u\}$, where b stands for the 'block' and u stands for the 'unblock' actions to model synchronous channels with data transfer delays: when a channel is blocked, it does not accept new I/O requests. ACA is the generalization of CA, and CA can be seen as ACA with one action type: data flow through a Reo node.

To reason about the correctness of ACA as semantic models for Reo, we introduce a *refinement* relation for ACA with various action types (in a specific case, for ACA and CA as the latter is equivalent to the ACA with one action: observation of dataflow on Reo ports). For simplicity we omit data constraints in the rest of this paper and focus on ACA synchronization constraints. First, we need to be able to say whether automata with various observable actions describe the same Reo circuit or not. Thus, we

introduce an action renaming operator to unify sets of action names, show its compositionality and then define a weak bisimulation relation to compare ACA with renamed actions. Proofs for the following properties can be found in [17].

Definition 3.2 (Action renaming) *For any $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$, let $\rho(\mathscr{A}, R)$ be an action renaming operator where R is a set of renamings in the form $x \rightarrow y, x \in H \subseteq \mathscr{N}, y = \rho(x), \rho : H \rightarrow H' \subseteq \mathscr{N}'$. $\rho(\mathscr{A}, R) = (S, \mathscr{N}', \rightarrow', s_0)$ is an ACA such that for any $(s, N, s') \in \rightarrow$ there exists $(s, N \setminus H \cup \rho[N \cap H], s'_0) \in \rightarrow'$.*

Proposition 3.1 (Commutativity of renaming and hiding) $\rho(\mathtt{hide}(\mathscr{A}, K), R) = \mathtt{hide}(\rho(\mathscr{A}, R), (K \setminus H) \cup \rho[K \cap H])$.

For $H \cap K = \emptyset$ it holds that $\rho(\mathtt{hide}(\mathscr{A}, K), R) = \mathtt{hide}(\rho(\mathscr{A}, R), K)$.

As in [18], we use the port synchronization function γ as follows: we write $\mathscr{N}_1'$ for $\mathscr{N}_1 \setminus \gamma_1[\mathscr{N}]$ and $\mathscr{N}_2'$ for $\mathscr{N}_2 \setminus \gamma_2[\mathscr{N}]$. If, for subsets $N_1 \subseteq \mathscr{N}_1$, $N_2 \subseteq \mathscr{N}_2$, it holds that $\gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$ we write $N_1 |_\gamma N_2 = (N_1 \cap \mathscr{N}_1') \cup \gamma_1^{-1}[N_1] \cup (N_2 \cap \mathscr{N}_2')$. Hence, $N_1 |_\gamma N_2$ is the union $N_1 \cup N_2$ but with the parts of N_1 and N_2 that are identified via γ_1 and γ_2 replaced by the shared names $\gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$. The following proposition states that the action renaming is compositional provided that in the product of ACA we rename the set of synchronized actions that is obtained from the sets of renamed actions in the original automaton:

Proposition 3.2 (Compositionality of action renaming) *Let $\mathscr{A}_1 = (S_1, \mathscr{N}_1, \rightarrow_{\mathscr{A}_1}, s_0^1)$ and $\mathscr{A}_2 = (S_2, \mathscr{N}_2, \rightarrow_{\mathscr{A}_2}, s_0^2)$ be two ACA with disjoint sets of action names, $\mathscr{N}_1 \cap \mathscr{N}_2 = \emptyset$. Let also $\gamma : \mathscr{N} \rightarrow \mathscr{N}_1 \times \mathscr{N}_2$ be an action synchronization function defined as $\gamma(n) = (\gamma_1(n), \gamma_2(n))$, where $\gamma_1 : \mathscr{N} \rightarrow \mathscr{N}_1$, $\gamma_2 : \mathscr{N} \rightarrow \mathscr{N}_2$ is a set of injective functions that map action names from the new set $\mathscr{N}$ into action names from the initial sets $\mathscr{N}_1$ and $\mathscr{N}_2$, $\mathscr{N} \cap (\mathscr{N}_1 \cup \mathscr{N}_2) = \emptyset$. Given sets of renamings $R_1 : \{x \rightarrow y = \rho_1(x), \rho_1 : H_1 \rightarrow L_1, H_1 \in \mathscr{N}_1\}$ and $R_2 : \{x \rightarrow y = \rho_2(x), \rho_2 : H_2 \rightarrow L_2, H_2 \in \mathscr{N}_2\}$, for $\mathscr{A}_1$ and $\mathscr{A}_2$, respectively, and a set of renamings $R = \{x \rightarrow y = \rho(x)\}$ for their product, where*

$$\rho(x) = \begin{cases} \rho_1(x) & x \in H_1 \subseteq H_1 |_\gamma H_2 \\ \rho_2(x) & x \in H_2 \subseteq H_1 |_\gamma H_2 \\ f(\rho_1(x_1), \rho_2(x_2)) & x = \gamma_1^{-1}(x_1) = \gamma_2^{-1}(x_2), x_1 \in H_1, x_2 \in H_2 \end{cases}$$

it holds that

$$\rho(\mathscr{A}_1 \bowtie_\gamma \mathscr{A}_2, R) = \rho(\mathscr{A}_1, R_1) \bowtie_\omega \rho(\mathscr{A}_2, R_2).$$

Here $\omega : \mathscr{M} \rightarrow \mathscr{M}_1 \times \mathscr{M}_2$, is an action synchronization function for the renamed actions defined as $\omega(n) = (\omega_1(n), \omega_2(n))$, $\omega_1 : \mathscr{M} \rightarrow \mathscr{M}_1$, $\omega_2 : \mathscr{M} \rightarrow \mathscr{M}_2$,

$$\mathscr{M} = \rho[H_1 |_\gamma H_2] \cup \mathscr{N} \setminus (H_1 |_\gamma H_2),$$
$$\mathscr{M}_1 = \rho[H_1] \cup \mathscr{N}_1 \setminus H_1, \quad \mathscr{M}_2 = \rho[H_2] \cup \mathscr{N}_2 \setminus H_2,$$
$$\gamma_1(n) = n_1 \wedge \gamma_2(n) = n_2 \text{ iff } \omega_1(\rho(n)) = \rho(n_1) \wedge \omega_2(\rho(n)) = \rho(n_2).$$

Note that hiding can be seen as renaming to unobservable action τ. Hence, it is compositional under the same conditions.

We define traces for ACA in a usual way: a finite or infinite sequence of transitions

$$r = s \xrightarrow{N_0} s_1 \xrightarrow{N_1} s_2 \xrightarrow{N_2} s_3 \ldots$$

is an s-trace in *ACA*. Let S^* be the set of all finite sequences over a set S. Given finite sequences σ_1 and σ_2, we denote their concatenation $\sigma_1 \cdot \sigma_2$. If for some ACA there exists an s-trace

$$s \xrightarrow{N_1} s_1 \xrightarrow{\emptyset} s_2 \xrightarrow{\emptyset} s_3 \xrightarrow{N_2} s_4 \xrightarrow{\emptyset} s_5 \xrightarrow{N_3} s_6 \xrightarrow{\emptyset} s',$$

where $N_1, N_2, N_3 \subseteq \mathscr{N}$ are sets of actions representing ACA labels, we write $s \xrightarrow{N_1 \cdot \emptyset \cdot \emptyset \cdot N_2 \cdot \emptyset \cdot N_3 \cdot \emptyset} s'$, or $s \xRightarrow{N_1 \cdot N_2 \cdot N_3} s'$ for the empty action set abstracted traces.

Definition 3.3 (Weak bisimulation) *Let $\mathscr{A} = (S, \mathscr{N}, \rightarrow, s_0)$ be an ACA. A weak bisimulation on $\mathscr{A}$ is an equivalence Θ on S such that for all $(s_1, s_2) \in \Theta$ if $s_1 \stackrel{N}{\Longrightarrow} p_1$ then $s_2 \stackrel{N}{\Longrightarrow} p_2$ for some $(p_1, p_2) \in \Theta$.*

States s_1 and s_2 are called weakly bisimilar iff there exists weak bisimulation $\mathscr{R}$ such that $(s_1, s_2) \in \mathscr{R}$.

Definition 3.4 (Action refinement) *Let $\mathscr{A} = (S, \mathscr{N}, \rightarrow_{\mathscr{A}}, s_0)$ and $\mathscr{B} = (Q, \mathscr{M}, \rightarrow_{\mathscr{B}}, q_0)$ be two ACA. We say that $\mathscr{A}$ is an action refinement of $\mathscr{B}$, written as $\mathscr{B} \preccurlyeq \mathscr{A}$, iff there exist a set $K \subseteq \mathscr{N}$ and a set of renamings $R = \{x \rightarrow y = \rho(x), \rho : \mathscr{N} \setminus K \rightarrow \mathscr{M}\}$ such that $\mathscr{B}$ and $\rho(\mathtt{hide}(\mathscr{A}, K), R) = (S, \mathscr{M}, \rightarrow', s_0)$ are weakly bisimilar.*

Proposition 3.3 (Compositionality of action refinement) *Let $\mathscr{A}_1 = (S_1, \mathscr{N}_1, \rightarrow_{\mathscr{A}_1}, s_0^1)$ and $\mathscr{B}_1 = (Q_1, \mathscr{M}_1, \rightarrow_{\mathscr{B}_1}, q_0^1)$ be two ACA such that $\mathscr{B}_1 \preccurlyeq A_1$ with a set of hidden actions $K_1 \subseteq \mathscr{N}_1$ and a set of renamings $R_1 : \{x \rightarrow y = \rho_1(x), \rho_1 : \mathscr{N}_1 \setminus K_1 \rightarrow \mathscr{M}_1\}$. Let $\mathscr{A}_2 = (S_2, \mathscr{N}_2, \rightarrow_{\mathscr{A}_2}, s_0^2)$ and $\mathscr{B}_2 = (Q_2, \mathscr{M}_2, \rightarrow_{\mathscr{B}_2}, q_0^2$ be two ACA such that $\mathscr{B}_2 \preccurlyeq \mathscr{A}_2$ with a set of hidden actions $K_2 \subseteq \mathscr{N}_2$ and a set of renaming $R_2 : \{x \rightarrow y = \rho_2(x), \rho_2 : \mathscr{N}_2 \setminus K_2 \rightarrow \mathscr{M}_2\}$. Assume also that $\mathscr{N}_1 \cap \mathscr{N}_2 = \mathscr{M}_1 \cap \mathscr{M}_2 = \emptyset$.*

The γ-synchronous product of automata $\mathscr{A}_1$ and $\mathscr{A}_2$ is the action refinement of the ω-synchronous product of automata $\mathscr{B}_1$ and $\mathscr{B}_2$, i.e.,

$$\mathscr{B}_1 \bowtie_\omega \mathscr{B}_2 \preccurlyeq \mathscr{A}_1 \bowtie_\gamma \mathscr{A}_2$$

with a set of hidden actions $K = K_1 |_\gamma K_2$ and a set of renamings $R = \{x \rightarrow y = \rho(x) : (\mathscr{N}_1 \setminus K_1) |_\gamma (\mathscr{N}_2 \setminus K_2)\} \rightarrow \mathscr{M}_1 |_\omega \mathscr{M}_2$ where $\rho(x)$ is defined as in Prop. 3.2, $\gamma = (\gamma_1, \gamma_2)$, $\gamma_1 : \mathscr{N} \rightarrow \mathscr{N}_1$, $\gamma_2 : \mathscr{N} \rightarrow \mathscr{N}_2$ and $\omega = (\omega_1, \omega_2)$, $\omega_1 : \mathscr{M} \rightarrow \mathscr{M}_1$, $\omega_2 : \mathscr{M} \rightarrow \mathscr{M}_2$ are action synchronization functions such that

$$\mathscr{N} \cap (\mathscr{N}_1 \cup \mathscr{N}_2) = \emptyset, \quad \mathscr{M} \cap (\mathscr{M}_1 \cup \mathscr{M}_2) = \emptyset, \text{ and}$$
$$\gamma_1(n) = n_1 \wedge \gamma_2(n) = n_2 \text{ iff } \omega_1(\rho(n)) = \rho(n_1) \wedge \omega_2(\rho(n)) = \rho(n_2).$$

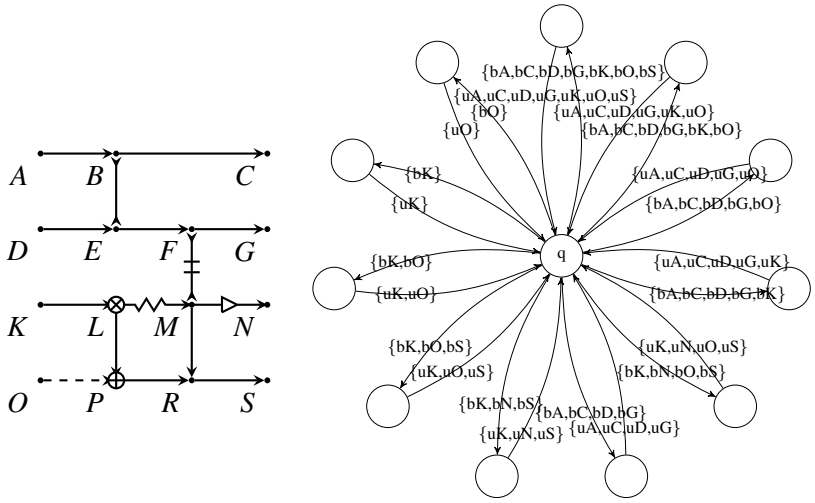
4 Semantic model for Reo implementation

Existing semantic models for Reo describe the coordination behavior of Reo circuits but do not show how to achieve it. We refer to the process of exchanging messages among remote Reo nodes in order to establish whether they are ready to accept or provide data as handshaking protocol: before transmitting service data, Reo nodes notify each other about their internal states. It is not clear how Reo nodes and channels should behave to determine which transitions are enabled and agree to perform one transition from the set of enabled ones. Specifying such behavior is important for the generation of executable coordination code [21].

In Reo, large synchronous regions can be constructed. By synchronous region we understand part of a circuit consisting of joint synchronous channels. Figure 2 shows a synchronous Reo circuit with eight boundary nodes and its operational semantics in the form of ACA. In this circuit, 4 source nodes

may provide data, 4 sink nodes may consume data, while 7 mixed internal nodes together with channels coordinate data flow. Here L is an exclusive *route* node, M is a *join* node, and other nodes and channels behave as explained in Section 2. Assume that a write request arrives on node A. A can accept this request iff (i) D has a pending write request, (ii) nodes C and G are ready to accept, and (iii) there is no data flow on port M (required by the semantics of the AsyncDrain channel). If nodes are deployed on remote machines, the node cannot decide whether to accept data until it receives messages from all parties it depends on. If several transitions are enabled (as shown by the ACA semantics, 11 transitions are possible in our example if all boundary nodes are ready to communicate), the implementation should non-deterministically choose one of them.

The goal of the handshaking protocol is to ensure that the internal choice made by Reo nodes locally lead to the correct implementation of the global observable behavior of the circuit. For example, if there are pending requests on nodes K, O and S, but no request on node N, the exclusive router L should transfer data to the node P and the merge node R should consume data from node P because transition $\{K,L,O,P,R,S\}$ (corresponds to an observable transition $\{K,O,P\}$) is enabled and should not be excluded by a local choice of a node or a channel with non-deterministic behavior.

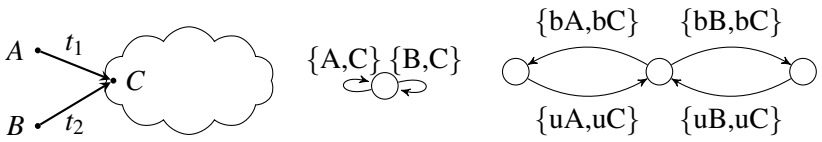


(a) Reo model (b) ACA

Figure 2: Complex synchronous circuit

The existing implementations for Reo either (i) decompose a circuit to synchronous regions and deploy all nodes and channels from such a region on a single machine [19], (ii) propagate technical messages to establish which nodes are ready to process data as if there were no delays [21]. The first approach is not a fully distributed solution: if the whole network consists of a single synchronous region, the coordination is performed by a single machine. The second approach implies no constraints on node deployment but it is not efficient for two reasons. Firstly, after an internal node receives a request message, it propagates it to the rest of the network. Meanwhile, if another message comes to the same node, it needs to be propagated again. Thus, internal nodes update their routing tables several times per execution cycle. Secondly, on each execution cycle, one node is elected to resolve non-deterministic choice and need to keep a list of enabled nodes for the whole synchronous region.

Consider a circuit with a merge fragment in Figure 3. Assuming that both input ports A and B receive write requests simultaneously, in the time-agnostic implementation [21], the merge node C propagates the request that arrives first. I.e., if $t_1 < t_2$, the merge node assumes that only transition $\{A,C\}$ is enabled.



(a) Circuit (b) CA (c) ACA with port blocking actions

Figure 3: Merge circuit and its semantics

In time $t_2 - t_1$ it learns that $\{B,C\}$ is enabled as well and sends a new technical message to notify other nodes in the synchronous region. This causes undesired traffic and does not scale well. Furthermore, the non-deterministic choice introduced by C is not resolved locally, but delegated to an external node (elected randomly, depending on its ID). Thus, the existing implementation is not optimal in time-aware environment and relies on centralized resolution of non-determinism at each execution cycle to ensure absence of inconsistencies.

We need a time-aware sematic model for Reo that displays their internal state changes (e.g., from *idle* to *waiting for reply* to *committed* and back to *idle*). TCA [2] models circuit time delays while ACA [16] allows multiple actions to be observed on node and channel ports. We combine these two models to describe our handshaking protocol.

Let $\mathscr{C}$ be a finite set of clocks and $v : \mathscr{C} \to \mathbb{R}_{\geq 0}$ be a clock assignment function as defined in [2]. Let also cc be a clock constraint for C, which is defined as a conjunction of atoms of the form $x \odot n$ where $x \in \mathscr{C}, \odot \in \{<, \leq, >, \geq, =\}$ and $n \in \mathbb{N}$. $CA(\mathscr{C})$ (or CA) denotes the set of all clock assignments and $CC(\mathscr{C})$ (or CC) the set of all clock constraints.

Definition 4.1 (Timed ACA) *A Timed ACA (TACA) is a tuple $\mathscr{A} = (S, \mathscr{C}, \mathscr{N}, \to, s_0, ic)$, where S is a finite set of states, $\mathscr{C}$ is a finite set of clocks, $\mathscr{N}$ is a set of action names derived from a set of port names $\mathscr{M}$ and a set of admissible action types $\mathscr{T}$, $\to \subseteq S \times 2^{\mathscr{N}} \times DC \times CC \times 2^{\mathscr{C}} \times S$ is a transition relation such that DC is the set of data constraints over a finite data domain Data, $s_0 \in S$ is an initial state, $ic : S \to CC$ is a function that assigns to any location s an invariance condition $ic(s)$.*

Let an injective function $act : \mathscr{M} \times \mathscr{T} \to \mathscr{N}$ define action names for each pair of a port name and an action type observed on the port as discussed in [16]. The action synchronization function $\gamma : \mathscr{N} \to \mathscr{N}_1 \times \mathscr{N}_2$ is defined through a pair of injective functions $\gamma_1 : \mathscr{N} \to \mathscr{N}_1$, $\gamma_2 : \mathscr{N} \to \mathscr{N}_2$ from a new set of action names $\mathscr{N}$ into $\mathscr{N}_1$ and $\mathscr{N}_2$ [18].

Definition 4.2 (Product of TACA) *For two TACA $\mathscr{A}_1 = (S_1, \mathscr{C}_1, \mathscr{N}_1, \to_1, s_0^1, ic_1)$ and $\mathscr{A}_2 = (S_2, \mathscr{C}_2, \mathscr{N}_2, \to_2, s_0^2, ic_2)$ and the action synchronization function $\gamma : \mathscr{N} \to \mathscr{N}_1 \times \mathscr{N}_2$ with $\gamma_1 : \mathscr{N} \to \mathscr{N}_1$ and $\gamma_2 : \mathscr{N} \to \mathscr{N}_2$, the TACA $\mathscr{A}_1 \bowtie_\gamma \mathscr{A}_2$, called the γ-synchronization product of $\mathscr{A}_1$ and $\mathscr{A}_2$, is given by $\mathscr{A}_1 \bowtie_\gamma \mathscr{A}_2 = (S_1 \times S_2, \mathscr{C}_1 \cup \mathscr{C}_2, \mathscr{N}_1' |_\gamma \mathscr{N}_2', \to, \langle s_0^1, s_0^2 \rangle, ic(\langle s_1, s_2 \rangle))$ where $ic(\langle s_1, s_2 \rangle) = \chi(ic(s_1)) \wedge \chi(ic(s_2))$, for any $s_1 \in S_1, s_2 \in S_2$, $\chi : CC_1 \cup CC_2 \to CC$ is a clock constraint and location invariance condition update function, and the transition relation $\to$ is determined by the following rules:*

$$\frac{s_1 \xrightarrow{N_1, g_1, cc_1}_1 t_1 \quad N_1 \subseteq \mathscr{N}_1'}{\langle s_1, s_2 \rangle \xrightarrow{N_1, g_1, \chi(cc_1)} \langle t_1, s_2 \rangle} \qquad \frac{s_2 \xrightarrow{N_2, g_2, cc_2}_1 t_2 \quad N_2 \subseteq \mathscr{N}_2'}{\langle s_1, s_2 \rangle \xrightarrow{N_2, g_2, \chi(cc_2)} \langle s_1, t_2 \rangle} \tag{1}$$

and

$$\frac{s_1 \xrightarrow{N_1, g_1, cc_1}_1 t_1 \quad s_2 \xrightarrow{N_2, g_2, cc_2}_1 t_2 \quad \gamma_1^{-1}(N_1) = \gamma_2^{-1}(N_2)}{\langle s_1, s_2 \rangle \xrightarrow{N_1 |_\gamma N_2, \gamma(g_1 \wedge g_2), \gamma(\chi(cc_1) \wedge \chi(cc_2))} \langle t_1, t_2 \rangle} \tag{2}$$

Function χ is introduced to update invariance conditions and clock constraints in composed circuits. In particular, this extension is needed to increase timeouts for composed networks. For example, if invariance conditions in $\mathscr{A}_1$ are in the form $x \leq T_1$ and the invariance conditions in $\mathscr{A}_2$ are in the form $x \leq T_2$, by setting $\chi(x \leq T_1) = x \leq T_1 + T_2$, and $\chi(x \leq T_2) = x \leq T_1 + T_2$, we extend timeout needed for the traversal of the composed network. Consequently, clock constraints in the form $x > T_1$ and $x > T_2$ are replaced with $x > T_1 + T_2$.

We define the hiding operator $\texttt{hide}(\mathscr{A}, K)$, where K is a non-empty set of actions $K \subseteq \mathscr{N}$, and a state-transition graph of TACA $G_{\mathscr{A}}$ analogously to the TCA [2]. The only distinction is that instead of the nodes set as in CA we deal with the action sets as in ACA.

To reason about time-agnostic Reo semantics, we need to abstract from time in TACA. We omit data constraints and focus on action synchronization constraints. Let $q = \langle s, v \rangle$ be a state of a state-transition graph of TACA $G_{\mathscr{A}}$. We call a finite or infinite sequence of transitions

$$r = q \xrightarrow{N_0,t_0} q_1 \xrightarrow{N_1,t_1} q_2 \xrightarrow{N_2,t_3} q_3 \ldots$$

a *q-trace* in $G_{\mathscr{A}}$. We say that a finite or infinite sequence of transitions

$$r = q \xrightarrow{N_0} q_1 \xrightarrow{N_1} q_2 \xrightarrow{N_2} q_3 \ldots$$

is an *untimed q-trace* iff there exist $t_0, t_1, t_2, \ldots \in \mathbb{R}_{\geq}$ such that

$$r' = q \xrightarrow{N_0,t_0} q_1 \xrightarrow{N_1,t_1} q_2 \xrightarrow{N_2,t_2} q_3 \ldots$$

is a q-trace in $G_{\mathscr{A}}$.

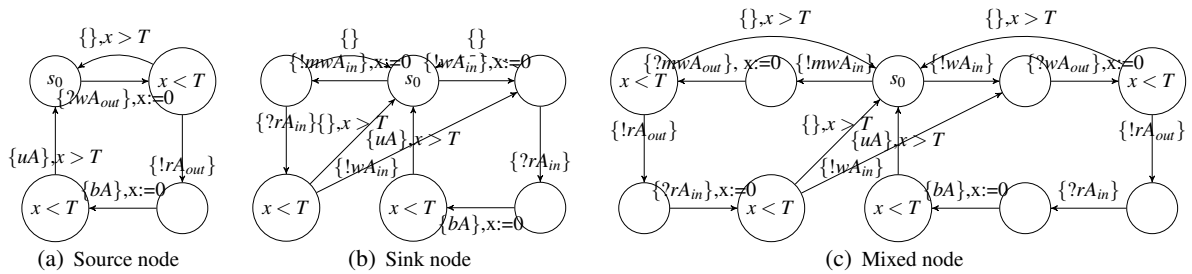
Let Q^* be the set of all finite sequences over a set $Q = \{\langle s, v \rangle \,|\, s \in S\}$. Given finite sequences σ_1 and σ_2, we denote their concatenation $\sigma_1 \cdot \sigma_2$. If for some TACA there exists an untimed trace $q \xrightarrow{N_1 \cdot \{\} \cdot \{\} \cdot N_2 \cdot \{\} \cdot N_3 \cdot \{\}} p$, where $N_1, N_2, N_3 \in 2^{\mathscr{N}}$ are sets of actions representing TACA labels, we write $q \xRightarrow{N_1 \cdot N_2 \cdot N_3} p$.

5 Reo handshaking protocol

I/O requests arrive to the Reo source nodes from *writers*, or to the sink nodes from *readers*. Writers and readers for synchronous regions are either external components or buffered Reo channels. Once a pending request is detected, the boundary node initiates handshaking message exchange through channels that connect it with its neighbors, reporting its status and requesting to confirm the ability to accept or provide data. At this stage, channels work as simple communication links between adjacent nodes regardless of their semantics.

Three message propagation strategies are possible: (i) *forward propagation*, when the handshaking is initiated by writers, readers remain passive and reply to the arrived messages; (ii) *backward propagation*, when the communication is initiated by readers, writers are passive; and (iii) *two-side propagation*, when both writers and readers can initiate the message exchange. Two-side propagation minimizes handshaking delay, but it is also more difficult to implement. In the remainder of this paper we consider forward propagation.

The handshaking behavior of Reo nodes depend on their type (input, output, simple internal, merge, replicate, route, or join) and (the number of) adjacent channels. Nodes can exchange three types of messages: (i) intention to write (*write*), (ii) possibility to read (*read*), and (iii) possibility to write (*may_write*).



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

Figure 4: Handshaking behavior of source, sink, and simple mixed nodes

The third message type is needed for routers to obtain status of nodes in alternative branches without giving definite promise to write data. Together with three message types, four actions are recognized at each channel port: (i) send a message to an adjacent node, (ii) receive a message from an adjacent node, (iii) block (or commit) port, (iv) unblock the port.

Figure 4 shows the TACA for simple Reo nodes: a *source* node with one output port A_{out}, a *sink* node with one input port A_{in}, and a *mixed* node with one input port A_{in} and one output port A_{out}. We use a set of action types $\mathscr{T}_1 = \{?,!\} \times \{w,r,mw\}$ to represent sending and receiving of *write*, *read*, and *may_write* messages, respectively. We also use a set of actions $\mathscr{T}_2 = \{b,u\}$ to define blocking and unblocking of node ports. Thus, a set of action names derived from the set of port names $\{A_{in}, A_{out}\}$ and a set of admissible action types $\mathscr{T} = \mathscr{T}_1 \cup \mathscr{T}_2$, is $\mathscr{N} = \mathscr{N}_{in} \cup \mathscr{N}_{out}$ where

$$\mathscr{N}_{in} = \{?wA_{in}, !wA_{in}, ?rA_{in}, !rA_{in}, ?mwA_{in}, !mwA_{in}, bA_{in}, uA_{in}\}$$

is a set of actions observed on the node's input port, and

$$\mathscr{N}_{out} = \{?wA_{out}, !wA_{out}, ?rA_{out}, !rA_{out}, ?mwA_{out}, !mwA_{out}, bA_{out}, uA_{out}\}$$

is the set of actions observed on its output port. If a node performs the same action on all its ports simultaneously, we write $\alpha A \,|\, \alpha \in \mathscr{T}$. For example, bA and uA stand for "block/unblock all ports of node A", respectively.

The handshaking behavior of a *source* node is shown in Figure 4(a)). The source node A sends a *write* message through its output port and waits for a reply for the time $x < T$, where T is a timeout large enough to guarantee that any message in the synchronous region of the circuit is propagated to the most remote (in terms of the communication delay) node and back. If the node does not receive a reply from the accepting party, it assumes that the latter is not ready to read and discards the request. If the answer is received (i.e., $!rA_{out}$), the node commits to transfer data by blocking its ports (bA). Finally, after the timeout expires, the node unblocks its ports (uA) and returns to the initial state.

The *sink* node (see Figure 4(b)) is a passive node that waits for the *write* or *may_write* messages. After such a message is received, it either confirms its readiness to accept ($?rA_{in}$) or ignores the message and returns to the initial state. The transition shown with the dashed line is not required for always accepting sink nodes. If the *write* message is received and the node is ready to accept, it goes to the committed state. If the *may_write* was received and the node confirmed the intention to accept, it awaits for the confirmation to write ($!wA_{in}$), replies to it and only then commits.

The *mixed* node exhibits the behavior which is the combination of the above. It accept the incoming messages and forwards the them to its neighbor though the output port ($?wA_{out}$ and $?mwA_{out}$), waits for the reply ($!rA_{out}$) and forwards it back through the input port ($?rA_{in}$). Similarly, if $!wA_{in}$ is received after $!mwA_{in}$, the mixed node forwards the *write* request to the output port, waits for the acknowledgement, forwards it to the sender, and commits.

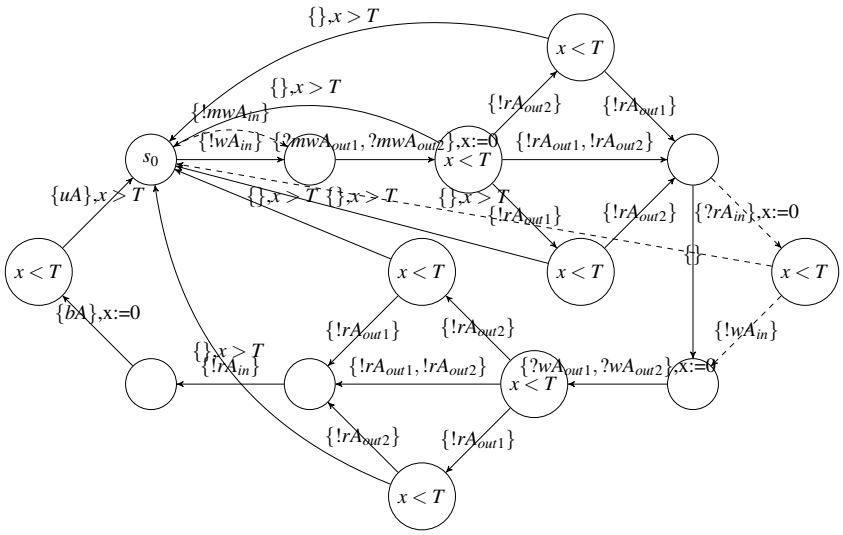


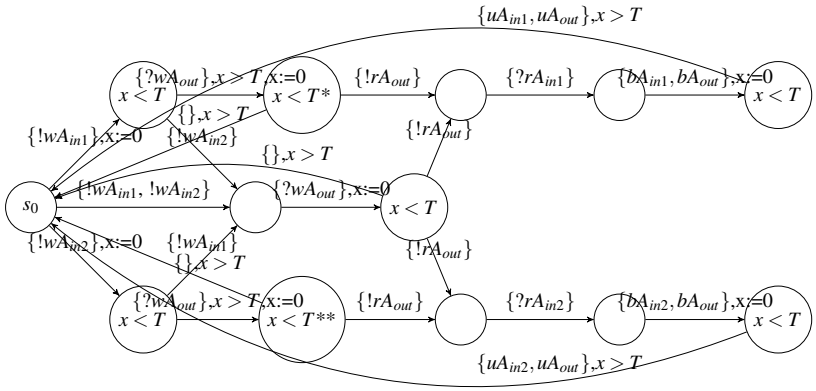
Figure 5: Replicate node

Note that it is important to acknowledge both *may_write* and *write* messages to be able to propagate I/O information in the circuit: the *read* message in response to a *may_write* message is just a confirmation that some accepting sink node exists in the circuit, yet the data transfer through a particular node may not happen due to the non-deterministic choices of internal Reo nodes. In contrast, the *read* message in response to a *write* message means that the exchanging parties agreed to transfer data.

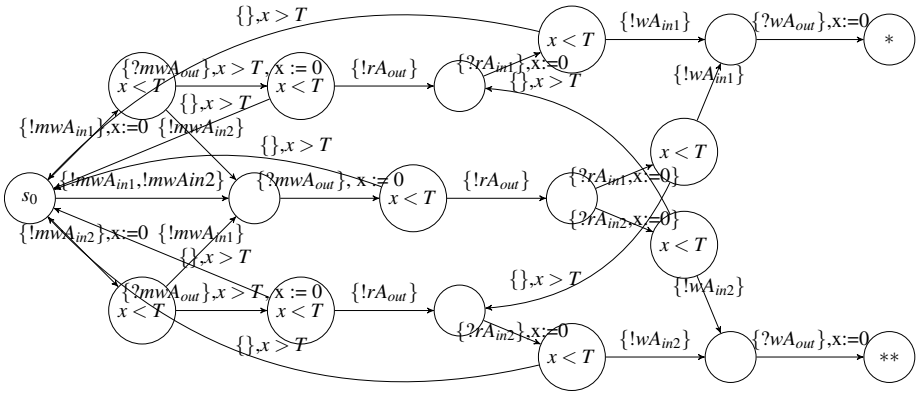
Considering the presence of *may_write* messages followed by *write* messages, T can be roughly estimated as C times the longest path in the (synchronous region of the) Reo circuit graph, where C is a constant that depends on the number of branches for merge nodes that may receive *may_write* messages. We will address the issue of computing the lower bound on timeout delays in our future work.

The handshaking behavior of the replicate node with input port A_{in} and output ports A_{out1} and A_{out2} is shown in Figure 5. Once a *write* or *may_write* message is received, the node sends a *may_write* message to its both output ports and waits for the replies. If meanwhile the timeout expires, the node returns to its initial state. If both neighbors confirm their ability to read, the further processing depends on the status of the input port: if the *write* message was received initially, the node A knows that it is able to provide data to its output ports and thus sends $\{?wA_{out1}, ?wA_{out2}\}$ to agree on the certain data exchange with them. If both $!rA_{out1}$ and $!rA_{out2}$ are observed, A forwards the reply back through its input port and commits (bA_{in}). Alternatively, if $!mwA_{in}$ initially triggered the decision making cycle on A, it has to request for the confirmation of the flow ($?rA_{in}$), and if it is confirmed ($!wA_{in}$), proceed as before. The difference in the procedure for the triggering *may_write* message is shown with dashed lines.

The behavior of the *merge* node with two input ports A_{in1} and A_{in2} and one output port A_{out} depends on the type of the incoming messages: both *write*, both *may_write*, or the combination of *write* and *may_write*. In the first case (see Figure 6(a)), the merge node in its initial state waits for an incoming message on at least one of its input ports, $!wA_{in1}$, $!wA_{in2}$ or both, forwards the incoming *write* message to the output port ($?wA_{out}$), and waits for the confirmation to read ($!rA_{out}$). If one input message is received, the node waits for the incoming message on the other input port. If the second message does not arrive within the timeout, it means that the other writer is not ready to write and A will proceed with the available request. If both input ports received a write request, the merge node chooses an incoming port to accept data from. Once the decision is made, the node A sends the *read* message to the selected input port ($?rA_{in1}$ or $?rA_{in2}$) and commits ($\{bA_{in1}, bA_{out}\}$ or $\{bA_{in2}, bA_{out}\}$).



(a) Merge node with both source ends ready to write



(b) Merge node with both source ends that may write

Figure 6: Merge node

In Figure 6(b), $!mwA_{in1}$ and $!mwA_{in2}$ are initially received. This means that the sender tries to establish which transactions are enabled. Consequently, after propagating the initial *may_write* message to the output port ($?mwA_{out}$) and receiving $!rA_{out}$, the node A needs to wait for the definitive confirmation to provide data from at least one of its input ends, $!wA_{in1}$ or $!wA_{in2}$. If such a message arrives, the node commits, otherwise, returns to the initial state. In the case when both input nodes are ready to write, A non-deterministically chooses one of the branches and sends the *read* message to it, either $?rA_{in1}$ or $?rA_{in2}$. If the selected input port receives the confirmation to write, the node forwards it to the output port and commits. However, the node that issued the initial *may_write* message may choose to fire a different transaction and the node A will not receive a *write* message. This should not exclude the transaction that involves the pending request on its second input port. Thus, on the timeout the node A sends the *read* message to the remaining source port. If the *write* message is finally received ($?wA_{out}$), the node forwards it to its output port and processes further messages as in Figure 6(a). Otherwise, due to the external choices, the node does not participate in the data transfer at this cycle and returns to its initial state. The combination of *write* and *may_write* messages yield an automaton with the behavioral patterns shown in the above two cases.

The *route* node with one input port A_{in} and two output ports A_{out1} and A_{out2} receives a *write* message $!wA_{in}$ or a *may_write* message $!mwA_{in}$, forwards it to its output ports and waits for the *read* messages.

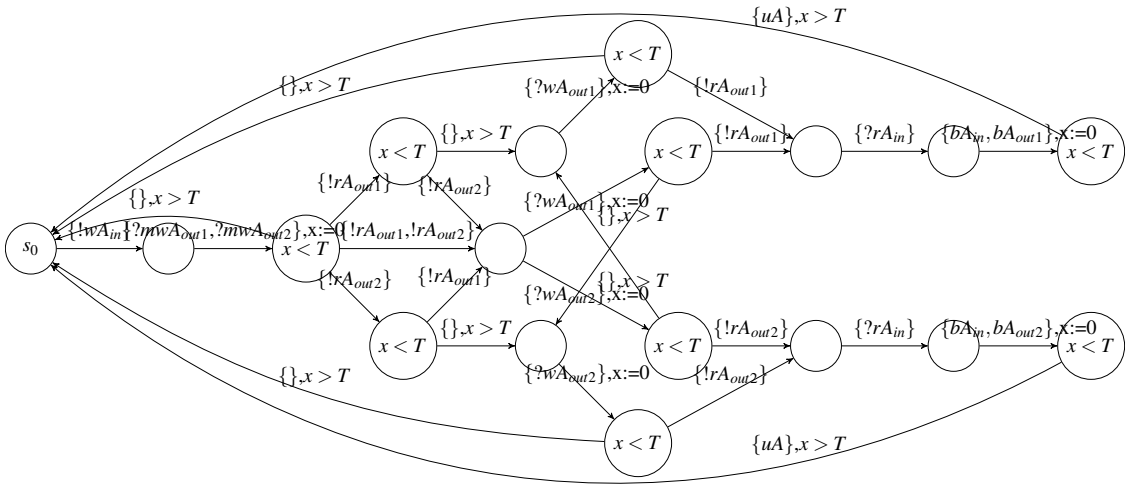


Figure 7: Route node

Figure 7 shows the case for the initial *write* message, the *may_write* is processed in a similar way and is omitted due to the lack of space. After at least one of the output ports confirms the ability to read ($!rA_{out1}$ and/or $!rA_{out2}$), the node A non-deterministically chooses among the available options and confirms the intention to write ($?wA_{out1}$ or $?wA_{out2}$). If the confirmation is received, the node proceeds by sending to the input port $?rA_{in}$ and commits ($\{bA_{in}, bA_{out1}\}$ or $\{bA_{in}, bA_{out1}\}$). Alternatively, on the expiration of the timeout, the route node tries to confirm the intention to write to another enabled output port if such an option is available.

For the initial *may_write* message the mechanism is similar, but before the node decides to issue a definite write message, it needs to confirm the ability to read $!rA_{in}$, receive the definite *write* message $?wA_{in}$, make a choice between $?wA_{out1}$ and $?wA_{out2}$, make sure that the port it had chosen acknowledged the ability to read (or try other options alternatively), send another confirmation to read to its input port ($!rA_{in}$) and commit.

For the *join* node A to commit, both its input ports A_{in1} and A_{in2} should receive *write* (or *may_write* with the consequent confirmation to write) messages, and its sink port should be able to read. If only one input request is received within the timeout, it is discarded. If both messages are *write* messages, the node A simply checks whether the sink end can read ($?wA_{out}$ followed by $!rA_{out}$), confirms the possibility of the flow to both senders ($\{!rA_{in1}, !rA_{in2}\}$) and commits.

Figure 8 shows a less obvious case with one *may_write* input message ($!mwA_{in1}$) and one *write* message ($!wA_{in2}$). The join node cannot guarantee the flow to its sink and thus senses whether the sink end can read with an uncertain *may_write* request ($?mwA_{out}$). If $!rA_{out}$ follows, the node should first request the uncertain input port to confirm the intention to write ($?rA_{in1}$). If so, the node behaves as in the previous case: it sends the *write* messages to its sink end, waits for the confirmation, forwards the confirmation to both input ports ($\{?rA_{in1}, ?rA_{in2}\}$) and commits.

Data flow in a Reo circuit is controlled not only by its nodes, but also by its channels that may perform operations on the data they receive. What is a channel and how do we implement them? Essentially, each channel is an abstraction for a set of hops in a computer network to and from a component that implements the channel's behavioral logic; its source port is known to data suppliers while its sink port is

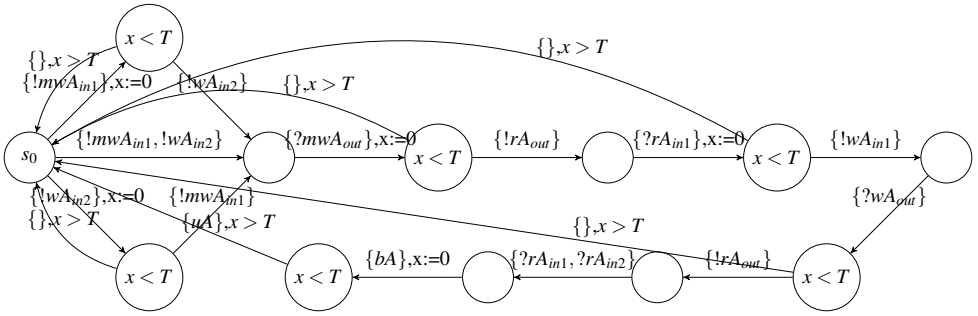
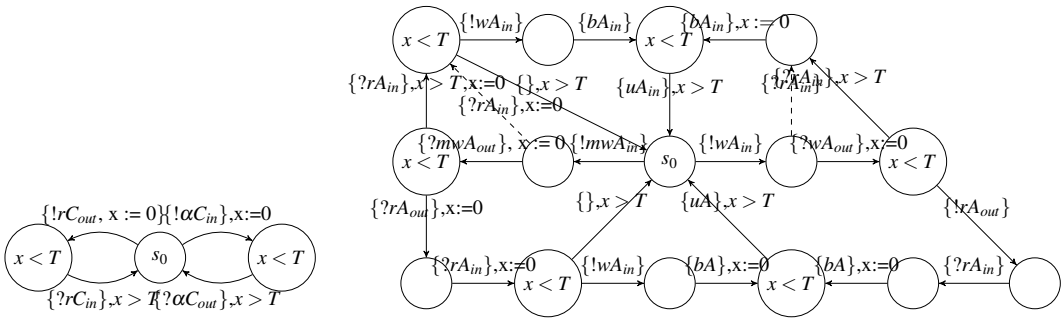


Figure 8: Join node



(a) Sync (b) LossySync

Figure 9: Handshaking behavior of Sync and LossySync channels

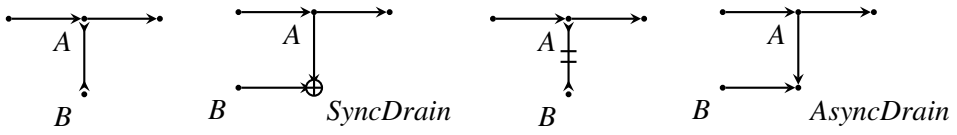
known to data consumers. From the viewpoint of handshaking protocol, Reo channels are communication links to exchange messages between adjacent nodes (e.g., Sync channel). However, channels behave also like nodes that determine which transitions are enabled (e.g., SyncDrain and AsyncDrain).

The handshaking behavior of a Sync channel c with communication delay t, source port C_{in} and sink port C_{out} can be modeled as shown in Figure 9(a). It accepts a *write* or a *may_write* message on its source end ($!\alpha C_{in}$ where $\alpha \in \{m, mw\}$) and notifies its sync end ($?\alpha C_{out}$). Similarly, it accepts a *read* message on its sink end ($!rC_{out}$) and transfers it to its source end ($?rC_{in}$).

The behavior of the LossySync channel as a transition link is analogous. However, its ability to lose data either when its output node is not ready to accept (context-dependent LossySync) or non-deterministically (context-independent LossySync) should be modeled explicitly. In the first case, the decision to accept data on the source end of the LossySync channel despite the fact that its sink end is unable to read can be modeled as shown in Figure 9(b). This figure shows the handshaking behavior of an always accepting mixed node that commits to a transaction that involves only its input port after the timeout of waiting for a *read* message from its output port expires. Thus, a context-dependent LossySync channel can be represented as a Sync channel joint to such a node.

The context-independent LossySync behaves similarly, but it may accept data without notifying its sink port, as shown by two dashed transitions $?bA_{in}$ in Figure 9(b), in right and left branches.

Alternatively, a non-deterministic LossySync can be modeled with the help of a router node that either chooses to pass data or reroute it to an unobservable always accepting output node τ (trash bin) where this data item is destroyed. In the same fashion, the behavior of SyncDrain and AsyncDrain channels can be modeled with the help of auxiliary join and merge nodes as shown in Figure 10.



(a) SyncDrain (b) SyncDrain (c) AsyncDrain (d) AsyncDrain

Figure 10: Modeling handshaking behavior of SyncDrain and AsyncDrain channels

6 Handshaking protocol correctness

In this section, we outline the sketch of a proof that the presented handshaking TACA provides correct implementation for Reo. To be able to show this formally, we need to define the notion of correct implementation.

Definition 6.1 (Observable trace) *For a synchronous Reo circuit with a set of nodes $\mathscr{P}$, let $\mathscr{A} = (S, \mathscr{C}, \mathscr{N}, \rightarrow_{\mathscr{A}}, s_0, ic)$, $\mathscr{N} = \mathscr{P} \times \mathscr{T}_{\mathscr{A}}$ be its handshaking TACA with port blocking, unblocking and auxiliary actions, $\{b,u\} \subset \mathscr{T}_{\mathscr{A}}$. We say that $s \xRightarrow{N_1 \cdot N_2 \cdot \ldots \cdot N_n} s'$ is an* observable *trace in $\mathscr{A}$ if it is an untimed trace in $\mathtt{hide}(\mathscr{A}, \mathscr{N} \setminus \mathscr{M})$ where $\mathscr{M} = \mathscr{P} \times \mathscr{T}_{\mathscr{B}}$, $\mathscr{T}_{\mathscr{B}} = \{b,u\}$.*

Definition 6.2 (Correct implementation) *For a synchronous Reo circuit with a set of nodes $\mathscr{P}$, let $\mathscr{B} = (Q, \mathscr{M}, \rightarrow_{\mathscr{B}}, q_0) \,|\, \mathscr{M} = \mathscr{P} \times \mathscr{T}_{\mathscr{B}}$, $\mathscr{T}_{\mathscr{B}} = \{b,u\}$ be its ACA, and $\mathscr{A} = (S, \mathscr{C}, \mathscr{N}, \rightarrow_{\mathscr{A}}, s_0, ic) \,|\, \mathscr{N} = \mathscr{P} \times \mathscr{T}_{\mathscr{A}}$ where $\mathscr{T}_{\mathscr{A}} \cap \mathscr{T}_{\mathscr{B}} = \{b,u\}$ be its handshaking TACA. The Reo handshaking protocol defined by $\mathscr{A}$ is a correct implementation of Reo iff there exists a mapping $\Theta : Q \rightarrow S$ such that*

- *for any $q \xrightarrow{M} q'$ in $\mathscr{B}$ there exists an observable trace $s \xRightarrow{N_1 \cdot N_2 \cdot \ldots \cdot N_n} s'$ in $\mathscr{A}$ such that $M = N_1 \cup N_2 \cup \ldots \cup N_n$,*
- *for any observable trace $s \xRightarrow{N_1 \cdot N_2 \cdot \ldots \cdot N_n} s'$ in $\mathscr{A}$, there exists $q \xrightarrow{M} q'$ in $\mathscr{B}$, $M = N_1 \cup N_2 \cup \ldots \cup N_n$.*

Intuitively, this definition requires the handshaking protocol to block and unblock only those sets of ports that appear in synchronization constraints of port blocking ACA. Since auxiliary actions may be required in the implementation, blocking and unblocking of all involved ports does not need to be simultaneous, it is sufficient that for each ACA transition to have a state in TACA with all necessary ports blocked and later unblock these ports.

To show the correctness of our protocol, we should (i) show that TACA provide correct implementation of port blocking ACA for each basic Reo node and channel as defined in [16], (ii) using product operator, define TACA for a composed circuit by synchronizing message exchange actions on shared ports and show that such a TACA is a correct implementation for the composed circuit.

For each Reo node and channel, we hide all handshaking messages in the time-abstracted version of TACA and show that the obtained automaton is weakly bisimilar to the port-blocking ACA for this node or channel (or equivalently, the initial TACA is the action refinement of the port blocking ACA). Compositionality of this relation helps to extend the proof to any Reo circuit. For a circuit with a set of nodes $\mathscr{P}$, its handshaking behavior is given by

$$\mathscr{A} = (S, \mathscr{C}, \mathscr{N}, \rightarrow_{\mathscr{A}}, s_0, ic),\ \mathscr{N} = \mathscr{P} \times \mathscr{T}_{\mathscr{A}},\ \mathscr{T}_{\mathscr{A}} = \{b, u, ?w, !w, ?mw, !mw, ?r, !r\}.$$

It can be checked that for all basic Reo channels and nodes, $\mathscr{B} \preccurlyeq \mathscr{A}$ with a set of hidden actions $K = \{\alpha \cdot X,\ \alpha \in \mathscr{T}_{\mathscr{A}} \setminus \mathscr{T}_{\mathscr{B}}, X \in \mathscr{P}\}$ and an empty set of renamings $R = \emptyset$. Since for each $q \xrightarrow{M} q'$ in $\mathscr{B}$ there exists $s \xRightarrow{M} s'$ in $\mathtt{hide}(\mathscr{A}, \mathscr{N} \setminus \mathscr{M})$, $\mathscr{A}$ is a correct implementation of $\mathscr{B}$.

To compose the TACA for handshaking, let us define a synchronization function $\gamma : (\gamma_1, \gamma_2)$, $\gamma_1 : \mathscr{N} \to \mathscr{N}_1, \gamma_2 : \mathscr{N} \to \mathscr{N}_2$ as follows. For any two joint ports A_{out} and B_{in} in a Reo circuit,

$$\begin{array}{rl} \gamma_1(?wA_{out}B_{in}) = ?wA_{out}, & \gamma_2(?wA_{out}B_{in}) = !wB_{in}, \\ \gamma_1(?mwA_{out}B_{in}) = ?mwA_{out}, & \gamma_2(?mwA_{out}B_{in}) = !mwB_{in}, \\ \gamma_1(!rA_{out}rB_{in}) = !rA_{out}, & \gamma_2(!rA_{out}rB_{in}) = ?rB_{in}, \\ \gamma_1(bA_{out}B_{in}) = bA_{out}, & \gamma_2(bA_{out}B_{in}) = bB_{in}, \\ \gamma_1(uA_{out}B_{in}) = uA_{out}, & \gamma_2(uA_{out}B_{in}) = uB_{in}. \end{array} \quad (3)$$

The definition of the implementation correctness of the handshaking protocol in the form of TACA is a weaker requirement than the action refinement and our synchronizing function enables an automaton for only one instance of suitable implementations. The TACA product without synchronizing blocking and unblocking actions in lines 4 and 5 of (3) will also yield a correct implementation for the corresponding port blocking ACA, but not the action refinement. It also generates a significantly larger automaton that is harder to deal with formally. However, in the distributed environment with communication delays it may be difficult to perform simultaneous port blocking and unblocking on remote nodes to signal that they are ready to transfer data. In practice, we are only interested in the existence of a state in which all ports involved into a firing transition are blocked. In the TACA product without synchronizing blocking and unblocking actions, Reo ports that are ready for firing transitions are blocked in any order. For example, in the case of a node with two ports, bA_{out} and bB_{in}, their interleaving $bA_{out}||bB_{in} = bA_{out}.bB_{in} + bB_{in}.bA_{out} + bA_{out}|bB_{in}$ gives us traces with the same set $\{bA_{out}, bB_{in}\}$ of performed actions, which conforms to our definition of the correct implementation.

7 Related work

Proença et al. [21] identifies five implementation approaches for Reo and offers their own framework, called Dreams. The most straightforward approach is a *speculative* approach: data is sent through the channels and rolled back when an inconsistency arises. This approach has never been implemented. The *automata-based* approach [19] relies on CA semantics and pre-computed behavior at compile time. Implementations based on connector *coloring* [19] compute all solutions for the behavior of each round and keep them in a routing table. Existing *search-based* implementations rely on structural operational semantics and are implemented in Alloy [15] and Maude [20]. The *constraint-based* approach [8] applies SAT solving techniques to search for single solutions on each round.

Dreams [21] is the first working implementation of a distributed engine for Reo. It is based on a *commit* and *send* message exchange and has common traits with our protocol. However, our protocol is a form of a speculative approach that does not require rollbacks and unnecessary data propagation. We provide a theoretical model for distributed Reo implementation with desired characteristics [21]:

- *Decoupling*: each node can be deployed on a separate machine, transitions in synchronous regions fire independently, and handshaking message exchange occurs within the boundaries of synchronous regions.
- *Scalability*: similarly to the Dreams, no global consensus is required. The behavior is computed per step, avoiding not only the state space explosion typical for the centralized implementation, but also the need for each node to remember the states of other nodes in the same synchronous region. In contrast to the Dreams, our implementation resolves non-determinism locally by each node or channel with inherent non-deterministic behavior. This is a more scalable approach because it does not use routing tables, the decision which transition to fire is taken in the distributed manner.

- *Reconfiguration*: runtime reconfiguration of a connector will not require any global or regional changes. Once a node detects a change in its environment, it only needs to adjust its own behavior. For example, if a new channel is connected to an input port of a simple node, this node becomes a merge node and should behave accordingly. Formal protocol adjustment rules to enable reconfiguration will be developed in our future work.
- *Fault tolerance*: with the exception of failures of committed nodes, our protocol is fault tolerant. The timeouts as opposed to the permanent locks in Dreams guarantee that a failure is treated as inability of a node to process data. Even in the case of a committed node failure, only current transaction is affected, at the next step (after timeout expires on each committed state) the remaining circuit resumes its normal operation.

In some applications of Reo such as coordination of processes in a multi-core execution environment, delays are negligible. Arriving requests are propagated through the network instantly and can be modeled using action synchronization. In [10], a Reo-to-C compiler which generates partially-distributed implementations for shared-memory platforms is presented. An optimization technique to improve the scalability of the generated code is later introduced [11]. In [9] and [14], the formal details of an implementation with partial-distribution based on synchronous regions are developed. An early Reo-to-Java compiler, which generates completely centralized implementations is described in [13] and used for service orchestration in [12].

The approach proposed in this paper can be used as foundation for distributed implementation of other coordination approaches for agent- and component-based systems where global consensus of synchronous entities is required. As opposed to commits and rollbacks typically employed to implement transactional processes [7], sensing messages like our *may_write* messages can be employed to elicit the states of remote entities before taking decisions locally.

8 Conclusions and Future Work

In this paper, we proposed a theoretical model for distributed implementation of Reo. We extended the definition of ACA with the notion of time, used this model to define what a correct implementation for Reo is, and described a handshaking protocol that complies with our definition of correct implementation. Our approach exhibits properties implied by Reo but not enforced in the previous implementations: the resolution of non-deterministic choices does not require centralized decisions, and the timeout mechanism ensures that the network does not lock due to a failure of a remote node or a channel. Consequently, this execution model is more suitable for real-time monitoring, QoS estimation, failure detection and reconfiguration.

Due to the space limitations we could not discuss some relevant aspects of the approach, most notably, computation of timeouts. We plan to extend the protocol to handle data constraints and implement a service that deploys and executes Reo circuits based on the presented model. The approach used in this paper can be applied to describe handshaking behavior of Reo nodes and channels assuming other propagation strategies: backward and two-side propagation.

References

[1] F. Arbab (2004): *Reo: A Channel-based Coordination Model for Component Composition*. *Mathematical Structures in Computer Science* 14, pp. 329–366, doi:10.1017/S0960129504004153.

[2] F. Arbab, C. Baier, F. de Boer & J. Rutten (2007): *Models and Temporal Logical Specifications for Timed Component Connectors*. Software and Systems Modeling 6(1), pp. 59–82, doi:10.1007/s10270-006-0009-9.

[3] F. Arbab, T. Chothia, R. van der Mei, M. Sun, Y.J. Moon & C. Verhoef (2009): *From Coordination to Stochastic Models of QoS*. In: Proc. COORDINATION 2009, LNCS 5521, Springer, pp. 268–287, doi:10.1007/978-3-642-02053-7_14.

[4] F. Arbab, T. Chothia, M. Sun & Y-J. Moon (2007): *Component Connectors with QoS Guarantees*. In: Proc. COORDINATION 2007, LNCS 4467, Springer, pp. 286–304, doi:10.1007/978-3-540-72794-1_16.

[5] F. Arbab, N. Kokash & M. Sun (2008): *Towards Using Reo for Compliance-aware Business Process Modelling*. In: Proc. ISoLA 2008, CCIS 17, Springer, pp. 108–123, doi:10.1.1.298.428.

[6] C. Baier, M. Sirjani, F. Arbab & J. Rutten (2006): *Modeling Component Connectors in Reo by Constraint Automata*. Science of Computer Programming 61(2), pp. 75–113, doi:10.1016/j.scico.2005.10.008.

[7] A. Baragatti, R. Bruni, H. Melgratti, U. Montanari & G. Spagnolo (2007): *Prototype platforms for distributed agreements*. ENTCS 180, pp. 21–40, doi:10.1016/j.entcs.2006.10.044.

[8] D. Clarke, J. Proença, A. Lazovik & F. Arbab (2011): *Channel-based coordination via constraint satisfaction*. Science of Computer Programming 76, doi:10.1016/j.scico.2010.05.004.

[9] S.-S. T. Q. Jongmans & F. Arbab (2013): *Global Consensus Through Local Synchronization*. In: Proc. FOCLASA 2012, 393, pp. 174 – 188, doi:10.1007/978-3-642-45364-9_15.

[10] S.-S. T. Q. Jongmans, S. Halle & F. Arbab (2013): *Reo: A Dataflow Inspired Language For Multicore*. In: Proc. DFM 2012, doi:10.1109/DFM.2013.14.

[11] S.-S. T. Q. Jongmans, S. Halle & F. Arbab (2014): *Automata-Based Optimization Of Interaction Protocols For Scalable Multicore Platforms*. In: Proc. COORDINATION 2014, LNCS 8459, Springer, pp. 65–82, doi:10.1007/978-3-662-43376-8_5.

[12] S.-S. T. Q. Jongmans, F. Santini, M Sargolzaei, F. Arbab & H. Afsarmanesh (2012): *Orchestrating web services using Reo: from circuits and behaviors to automatically generated code*. In: Proc. ESOCC 2012, LNCS 7592, Springer, pp. 1 – 16, doi:10.1109/PDP.2014.19.

[13] Sung-Shik Jongmans & Farhad Arbab (2013): *Modularizing and Specifying Protocols among Threads*. In: Proc. PLACES 2012, EPTCS 109, pp. 34–45, doi:10.4204/eptcs.109.6.

[14] Sung-Shik T. Q. Jongmans, Francesco Santini & Farhad Arbab (2014): *Partially-Distributed Coordination with Reo*. In: Proc. PDP 2014, IEEE, pp. 697–706, doi:10.1109/PDP.2014.19.

[15] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi & H. Iravanchi (2008): *Modeling and analysis of Reo connectors using Alloy*. In: Proc. COORDINATION 2008, LNCS 5052, pp. 169–183, doi:10.1007/978-3-540-68265-3_11.

[16] Changizi B. Kokash, N. & F.: Arbab (2010): *A Semantic Model for Service Composition with Coordination Time Delays*. In: Proc. ICFEM'10, LNCS, Springer, doi:10.1007/978-3-642-16901-4_9.

[17] N. Kokash (2014): *Handshaking Protocol for Distributed Implementation of Reo*. Technical Report TR 2014-01, LIACS, Leiden University.

[18] N. Kokash, C. Krause & E.P. de Vink (2010): *Data-Aware Design and Verification of Service Composition with Reo and mCRL2*. In: Proc. of SAC 2010, ACM Press, pp. 2406–2413, doi:10.1145/1774088.1774590.

[19] Z. Maraikar, A. Lazovik & F. Arbab (2008): *Building mashups for the enterprise with SABRE*. In: Proc. ICSOC 2008, LNCS 5364, pp. 70–83, doi:10.1007/978-3-540-89652-4_9.

[20] M. Mousavi, M. Sirjani & F. Arbab (2006): *Formal semantics and analysis of component connectors in Reo*. ENTCS 154(1), pp. 83–99, doi:10.1016/j.entcs.2005.12.034.

[21] Jose Proença, Dave Clarke, Erik de Vink & Farhad Arbab (2012): *Dreams: a framework for distributed synchronous coordination*. In: Proc. SAC 2012, ACM, doi:10.1145/2245276.2232017.