Decoupled execution of synchronous coordination models via behavioural automata

José Proença Dave Clarke
IBBT-DistriNet, KUL,
Leuven, Belgium
{jose.proenca,dave.clarke}@cs.kuleuven.be

Erik de Vink
TUE, Eindhoven,
The Netherlands
evink@win.tue.nl

Farhad Arbab
CWI, Amsterdam,
The Netherlands
farhad.arbab@cwi.nl

Synchronous coordination systems allow the exchange of data by logically indivisible actions involving all coordinated entities. This paper introduces behavioural automata, a logically synchronous coordination model based on the Reo coordination language, which focuses on relevant aspects for the concurrent evolution of these systems. We show how our automata model encodes the Reo and Linda coordination models and how it introduces an explicit predicate that captures the concurrent evolution, distinguishing local from global actions, and lifting the need of most synchronous models to involve all entities at each coordination step, paving the way to more scalable implementations.

1 Introduction

Synchronous constructs in languages such as Reo [1] and Esterel [7] are useful for programming reactive systems, though in general their realisations for coordinating distributed systems become problematic. For example, it is not clear how to efficiently implement the high degrees of synchronisation expressed by Reo in a distributed context. To remedy this situation, the GALS (globally asynchronous, locally synchronous) model [9, 13] has been adopted, whereby local computation is synchronous and communication between different machines is asynchronous.

Our work contributes to the field of coordination, in particular to the Reo coordination language, by incorporating the same ideas behind GALS in our approach to execute synchronisation models. More specifically, we introduce *behavioural automata* to model synchronous coordination, inspired in Reo [6]. Each step taken by an automata corresponds to a round of "synchronous" actions performed by the coordination layer, where data flow atomically through a set of points of the coordinated system. The main motivation behind behavioural automata is to describe the synchronous semantics underlying Dreams [18], a prototype distributed framework briefly discussed in §5.2 that stands out by the decoupled execution of Reo-like coordination models in a concurrent setting. Dreams improves the performance and scalability of previous attempts to implement similar coordination models. Our automata model captures exactly the features implemented by Dreams.

Behavioural automata assume certain properties over their labels, such as the existence of a composition operator, and use a predicate associated to each of its states that is needed to guide the composition of automata. Different choices for the composition operator of labels and the predicates yield different coordination semantics. We instantiate our automata with the semantics for Reo and Linda coordination models, but other semantic models can also be captured by our automata [18]. We do not instantiate behavioural automata with Esterel as the propagation of synchrony in this language differs from our dataflow-driven approach [3].

Summarising, the main contributions of this paper are:

- a *unified* automata model that captures dataflow-oriented synchronous coordination models;

Submitted to:
FOCLASA 2011

© J. Proença, D. Clarke, E. de Vink & F. Arbab
This work is licensed under the
Creative Commons Attribution License.

- the introduction of *concurrency predicates*, increasing the expressiveness of the model when dealing with composed automata; and
- the *decoupling of execution* of a distributed implementation based on our automata model, by avoiding unnecessary synchronisation of actions whenever possible.

Each behavioural automaton has a concurrenty predicate that indicates, for each state, which labels of other automata require synchronisation. When composing two automata, labels must be either composed in a pairwise fashion, or they can be performed independently when the concurrency predicate does not require synchronisation. We exploit how to use concurrency predicates to distinguish transitions of a composed automaton that originate from all intermediate automata, or from only a subset of them. We also illustrate how to obtain more complex notions of coordination by increasing the complexity of concurrency predicates.

This paper is organised as follows. We introduce behavioural automata in §2. We then encode Reo as behavioural automata in §3 and Linda as behavioural automata in §4. In §5 we motivate the need for concurrency predicates, both from a theoretical and practical perspectives. We conclude in §6.

2 A stepwise coordination model

In this section we present an automata model, dubbed *behavioural automata*. This model represents our view of a dataflow-driven coordination system, following the categorisation of Arbab [3]. Each transition in an automaton represents the *atomic* execution of a number of actions by the coordination system. We describe the behaviour of a system by the *composition* of the behaviour of its sub-systems running concurrently, each with its own automaton. Furthermore, we allow the *data values* exchanged over the coordination layer to influence the choice of how components communicate with each other as well. We borrow ideas from the Tile model [14, 4], distinguishing evolution in time (execution of the coordination system) and evolution in space (composition of coordination systems). Behavioural automata can be built by *composing* more primitive behavioural automata, and each transition of an automaton denotes a round of the coordination process, where data flow *atomically* through zero or more ports of the system.

We use behavioural automata to give semantics to Reo, based on the constraint automata model [6], and to (distributed) Linda [15]. Each label of an automaton describes which ports should have dataflow, and what data should be flowing in each port. We write $\mathbb{P}$ to denote a global set of ports, $\mathsf{L}[P]$ to denote the set of all labels over the ports $P \subseteq \mathbb{P}$, and $\mathbb{D}$ to denote a global set of data values. We associate a predicate over labels to each state q of an automaton, referred to as $\mathcal{C}(q)$. These predicates are used to guide the composition of behavioural automata.

Definition 1 (Behavioural automata) *A* behavioural automaton *of a system over a set of ports $P \subseteq \mathbb{P}$ is a labelled transition system $\langle Q, \mathsf{L}[P], \rightarrow, \mathcal{C} \rangle$, where $\mathsf{L}[P]$ is the set of labels over P, $\rightarrow \subseteq Q \times \mathsf{L}[P] \times Q$ is the transition relation, and $\mathcal{C} : Q \rightarrow \mathbf{2}^{\mathsf{L}[P]}$ is a predicate over states and labels, called* concurrency predicate*, regarded as a function that maps states to sets of labels.*

The key ingredients of behavioural automata are *atomic steps* and *concurrency predicates*. Each label of a behavioural automaton has an associated atomic step, which captures aspects such as the ports that have flow and the data flowing through them, and concurrency predicate describe, for each state, which labels from other automata running concurrently require synchronisation.

Example 1 (Alternating coordinator) *We present the* alternating coordinator *(AC) in Figure 1. It receives data from two data writers W_1 and W_2, and sends data to a reader R. The components W_1, W_2*

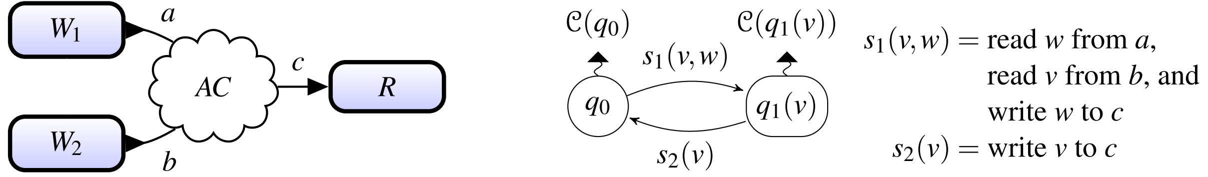


Figure 1: Alternating coordinator (left), and its behavioural automaton (right).

and R are connected, respectively, to the ports a, b and c of the alternating coordinator. The alternating coordinator describes how data can flow between the components, and coordination is specified by the behavioural automaton depicted on the right side of Figure 1. Each transition of this automaton represents a possible step in time of the coordinator AC, describing how the ports a, b, and c can have dataflow. Initially, the coordinator is in state q_0, where the only possible action is reading a value w from W_1 through a and sending it to the reader R through c, while reading and buffering a value v sent by W_2 through b. Note that if only one of the writers can produce data, the step cannot be taken, and the system cannot evolve. In the next state, q_1, the only possible step is to send the value v to the reader R, returning to state q_0. The arrows between states represent the transition relation $\rightarrow$. In both states there is the possibility of allowing the concurrent execution of other automata, provided that this execution does not interfere with the current behaviour. The conditions of when other automata can execute concurrently are captured by the concurrency predicate $\mathcal{C}$, depicted by squiggly arrows ($\rightsquigarrow$) from each state.

2.1 Labels, atomic steps and concurrent predicates

Labels over a set of ports P are elements from a set $\mathsf{L}[P]$ with some properties required for composition, which we will introduce later. Furthermore, a label $\ell \in \mathsf{L}[P]$ can be restricted to a smaller set of ports $P' \subseteq P$, written $\ell^{(P')}$. We require each label $\ell \in \mathsf{L}[\mathbb{P}]$ to have an associated description of where and which data flow in the connector, written as $\alpha(\ell)$, and captured by the notion of *atomic step*.

Definition 2 (Atomic step) *An* atomic step *over the alphabet $P \subseteq \mathbb{P}$ is a tuple $\langle P, F, IP, OP, data \rangle$ where:*

$$F \subseteq P \qquad IP \subseteq F \qquad OP \subseteq F \qquad IP \cap OP = \emptyset \qquad \text{and} \qquad data : (IP \cup OP) \rightarrow \mathbb{D}.$$

We write $\mathsf{AS}[P]$ to denote the set of all atomic steps over the ports in P. P is a set of ports in the scope of the atomic step. The flow set F is the set of ports that *synchronise*, i.e., that have data flowing in the same atomic step. The sets IP and OP represent the input and output ports of the atomic step that have dataflow, and whose values are considered to be relevant when performing a step. Ports in F but not in IP or OP are ports with dataflow, but whose data values are not relevant, that is, they are used only for imposing synchronisation constraints. The data values that flow through the relevant ports are given by the data function *data*. We distinguish IP and OP to capture data dependencies.

Concurrency predicates are used to compose behavioural automata. When composing two automata a_1 and a_2, if a_1 has ports P_1, has the concurrency predicate $\mathcal{C}_1$, and is in state q_1, then $\ell_2^{(P_1)} \in \mathcal{C}_1(q_1)$ means that a_2 can perform ℓ_2 only when composed with a transition from a_1, otherwise a_2 can perform ℓ_2 without requiring a_1 to perform a transition.[1] When clear from context, we omit the restriction and write $\ell_2 \in \mathcal{C}_1(q_1)$ instead of $\ell_2^{(P_1)} \in \mathcal{C}_1(q_1)$. We give a possible definition for concurrency predicates based

[1] We present a variation of the original definition of concurrency predicates [18] to make the decision of belonging to a concurrent predicate more local.

solely on the set of known ports.[2] Given a connector with known ports P_0, the concurrency predicate of every state is given by the predicate

$$cp(P_0) \;=\; \{\ell \mid \alpha(\ell) = \langle P,F,IP,OP,data\rangle\,, P_0 \cap F \neq \emptyset\}\,. \tag{1}$$

Example 2 *We define the atomic steps and concurrency predicates from Example 1 as follows.*

$$\alpha(s_1(v,w)) \;=\; \langle P, abc, ab, c, \{a,b,c \mapsto w,v,w\}\rangle \qquad \mathcal{C}(q_1(v)) \;=\; cp(P)$$
$$\alpha(s_2(v)) \;=\; \langle P,\;\; c,\;\; \emptyset, c, \{c \mapsto v\}\rangle \qquad \mathcal{C}(q_0) \;=\; cp(P)$$

For simplicity, we write $a_1 \dots a_n$ instead of $\{a_1,\dots,a_n\}$ when the intended notion of set is clear from the context. The alphabet P is $\{a,b,c\}$, and the concurrency predicates allow only steps where none of the known ports has flow.

2.2 Composition of behavioural automata

To compose behavioural automata we require labels to be elements of a partial monoid $\langle \mathsf{L},\otimes\rangle$, that is, (1) there must be a commutative operator $\otimes : \mathsf{L}^2 \rightharpoonup \mathsf{L}$ for labels, and (2) the composition of two labels can be undefined, meaning that they are incompatible. For technical convenience, we require $\otimes$ to be associative and to have an identity element. The atomic step $\langle P,F,IP,OP,data\rangle$ of a composed label $\ell_1 \otimes \ell_2$ must obey the following conditions, where, for every label ℓ_1 or ℓ_2, $\alpha(\ell_i) = \langle P_i,F_i,IP_i,OP_i,data_i\rangle$.

$$P \subseteq P_1 \cup P_2 \qquad IP \subseteq (IP_1 \cup IP_2)\backslash(OP_1 \cup OP_2) \qquad data_1 \frown data_2$$
$$F \subseteq F_1 \cup F_2 \qquad OP \subseteq OP_1 \cup OP_2 \qquad data = data_1 \cup data_2$$

The atomic step of a label ℓ is represented by $\alpha(\ell)$. The notation $m_1 \frown m_2$ represents that the values of the common domain of mappings m_1 and m_2 match. The requirements on the sets IP and OP reflect that when composing two atomic steps, the input ports that have an associated output port are no longer treated as input ports (since the dependencies have been met), and the output ports are combined. The intuition behind the removal of input ports that match an output port is the preservation of the semantics of Reo: multiple connections to an output port replicate data, but multiple connections to input data require the merging of data from a single source.

We now describe the composition of behavioural automata based on the operator $\otimes$ and on concurrency predicates. This composition mimics the composition of existing Reo models [6, 11, 8].

Definition 3 (Product of behavioural automata) *The product of two behavioural automata $b_1 = \langle Q_1, \mathsf{L}[P_1], \rightarrow_1, \mathcal{C}_1\rangle$ and $b_2 = \langle Q_2, \mathsf{L}[P_2], \rightarrow_2, \mathcal{C}_2\rangle$, denoted by $b_1 \bowtie b_2$, is the behavioural automaton $\langle Q_1 \times Q_2, \mathsf{L}[P_1 \cup P_2], \rightarrow, \mathcal{C}\rangle$, where $\rightarrow$ and $\mathcal{C}$ are defined as follows:*

$$\rightarrow \;=\; \{\langle (p,q),\ell,(p',q')\rangle \mid p \xrightarrow{\ell_1}_1 p', q \xrightarrow{\ell_2}_2 q', \ell = \ell_1 \otimes \ell_2, \ell \neq \bot\} \;\cup \tag{2}$$

$$\{\langle (p,q),\ell,(p',q)\rangle \mid p \xrightarrow{\ell}_1 p', \ell^{(P_2)} \notin \mathcal{C}_2(q)\} \;\cup\; \{\langle (p,q),\ell,(p,q')\rangle \mid q \xrightarrow{\ell}_2 q', \ell^{(P_1)} \notin \mathcal{C}_1(p)\} \tag{3}$$

$$\mathcal{C}(p,q) \;=\; \mathcal{C}_1(p) \cup \mathcal{C}_2(q) \quad \textit{for} \quad p \in Q_1, q \in Q_2. \tag{4}$$

Case (3) covers the situation where one of the behavioural automata performs a step admitted by the concurrency predicate of the other, and case (4) defines the composition of two concurrency predicates.

In practice, our framework based on behavioural automata, briefly described in §5.2, uses a symbolic representation for data values assuming that variables can be instantiated after selecting the transition. This suggests the use of a late-semantics for data-dependencies. Our approach to compose labels resembles Milner's synchronous product in SCCS [17], with the main difference that the product of behavioural automata do not require the all labels to be synchronised. The product of labels from two behavioural automata can be undefined, and labels can avoid synchronisation when the concurrency predicate holds.

[2] Other semantic models may require more complex concurrency predicates. For example, the concurrency predicates for the Reo automata model [8] depend on the current state (Section 3.6.2 of [18]).

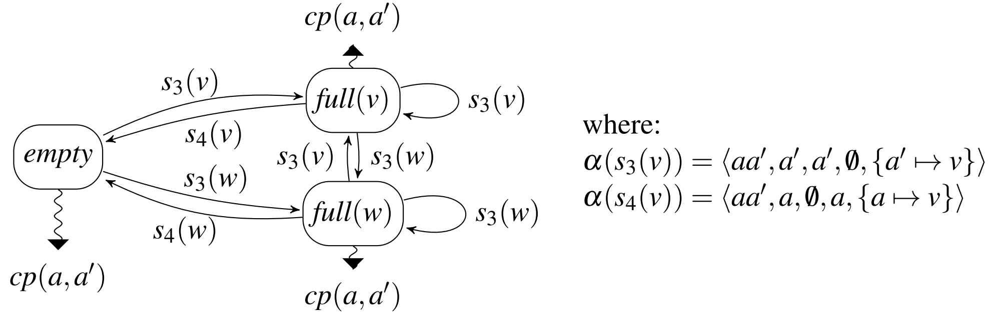


Figure 2: Behavioural automaton of the lossy-FIFO connector.

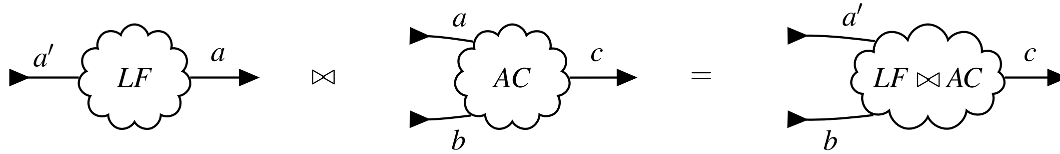


Figure 3: The sink and source ports of *LF*, *AC*, and their composition.

2.3 Example: lossy alternator

Recall the behavioural automaton *AC* of the alternating coordinator, illustrated in Figure 1. Data is received always via ports a and b simultaneously, and sent via port c, alternating the values received from a and b. We now imagine the following scenario: the data on a becomes available always at a much faster rate than data on b. To adapt our alternating coordinator to this new scenario, we introduce a lossy-FIFO connector *LF* [1] and compose it with the alternating coordinator, yielding $LF \bowtie AC$.

Recall the definition of $cp : \mathbb{P} \to \mathsf{L}[\mathbb{P}]$ given by Equation (1). The behavioural automaton for the lossy-FIFO connector is depicted in Figure 2, and its atomic steps range over the ports $\{a, a'\}$, where a' is an input port and a is an output port. We depict the interface of both of these connectors on left hand side of Figure 3. After combining the behavioural automata of the two connectors, they become connected via their shared port a. The new variation of the alternating coordinator can then be connected to data producers and consumers by using the ports a', b and c, as depicted at the right hand side of Figure 3.

Intuitively, the lossy-FIFO connector receives data a' and buffers its value before sending it through a. When the buffer is full data received from a' replaces the content of the buffer. The connector resulting from the composition $LF \bowtie AC$ is formalised in Table 1 and in Figure 4. The flow sets of the labels $s_1(v,w)$, $s_2(v)$, $s_3(v)$ and $s_4(v)$ are, respectively, abc, c, a', and $a'a$, and the set of known ports is $P = \{a', a, b, c\}$. Let $\mathcal{C}_{LF}$ and $\mathcal{C}_{AC}$ be the concurrency predicates of *LF* and *AC*. The concurrency predicate $\mathcal{C}_{LF \bowtie AC}$ for $LF \bowtie AC$ results from the union of the predicates of the states of each behavioural automaton, and corresponds precisely to the concurrency predicate that maps each state to $cp(a', a, b, c)$. The name of each state in $LF \bowtie AC$ is obtained by pairing names of a state from *LF* and a state from *AC*. Some states and transitions are coloured in grey with their labels omitted to avoid cluttering the diagram.

From the diagram it is clear that some transitions originate only from the *LF* or the *AC* connector, while others result from the composition via the operator $\otimes$. The transitions $s_2(v)$ and $s_3(w)$ can be per-

$\otimes$	$s_1(u,v)$	$s_2(w)$
$s_3(y)$	$\bot$	$\langle P, a'c, a', c, \{a', c \mapsto y, w\}\rangle$
$s_4(z)$	$\bot$ (for $z \neq v$)	$\bot$
$s_4(v)$	$\langle P, abc, ab, c, \{a, b, c \mapsto v, u, v\}\rangle$	$\bot$

LF	$\mathcal{C}_{LF}(empty)$	$\mathcal{C}_{LF}(full(v'))$
$s_1(v,w)$	*true*	*true*
$s_2(v)$	*false*	*false*

AC	$\mathcal{C}_{AC}(q_0)$	$\mathcal{C}_{AC}(q_1(v'))$
$s_3(v)$	*false*	*false*
$s_4(v)$	*true*	*true*

Table 1: Atomic steps of the composition of labels from LF and AC (left), and verification of the concurrency predicate for each label (right).

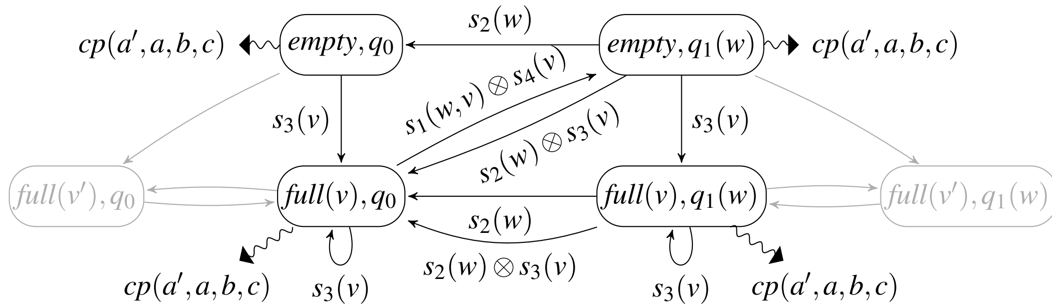


Figure 4: Behavioural automaton for the composition of LF and AC.

formed simultaneously or interleaved; simultaneously because $s_2(v) \otimes s_3(w)$ is defined, and interleaved because $\mathcal{C}_{LF}$ never contains $s_2(v)$ and $\mathcal{C}_{AC}$ never contains $s_3(w)$. The possible execution scenarios of these atomic steps follow our intuition that steps 'approved' by concurrency predicates can be performed independently. The steps $s_1(u,v)$ and $s_4(w)$ can be taken only when composed.

2.4 Locality

We introduce the notion of locality as a property of behavioural automata that guarantees the absence of certain labels in the concurrency predicates of *independent* behavioural automata, that is, in automata without shared ports.

Definition 4 (Locality of behavioural automata) *A behavioural automaton $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{C}\rangle$ obeys the* locality property *if, for any port set P' such that $P \cap P' = \emptyset$, $\forall \ell \in \mathsf{L}[P'] \cdot \forall q \in Q \cdot \ell^{(P)} \notin \mathcal{C}(q)$.*

Any two behavioural automata with disjoint port sets that obey the locality property can therefore evolve concurrently in an interleaved fashion. Let $b = b_1 \bowtie b_2$ be a behavioural automaton and ℓ a label from b_1. We say ℓ is a *local step* of b if $(q_1,q_2) \xrightarrow{\ell} (q_1',q_2')$ is a transition of b and either $q_1 \xrightarrow{\ell}_1 q_1'$, $q_2 = q_2'$, and $\ell \in \mathcal{C}_2(q_2)$; or $q_2 \xrightarrow{\ell}_2 q_2'$, $q_1 = q_1'$, and $\ell \in \mathcal{C}_1(q_1)$. In the behavioural automaton exemplified in Figure 4, the local steps are exactly the transitions labelled by the steps $s_2(w)$ and $s_3(v)$.

Proposition 1 *Let $b = b_1 \bowtie b_2 \bowtie b_3$ be a behavioural automaton where $b_i = \langle Q_i, \mathsf{L}[P_i], \rightarrow_i, \mathcal{C}_i \rangle$, for $i \in 1..3$, and assume the locality property from Definition 4 holds for b_1, b_2 and b_3. Suppose $P_1 \cap P_3 = \emptyset$. Then, for any step $\ell_1^{(P_1)} \in \mathsf{L}[P_1]$ performed by b_1 and $q_2 \in Q_2$, if $\ell_1^{(P_2)} \notin \mathcal{C}_2(q_2)$ then ℓ_1 is a local step of b.*

Proof. Observe that $\bowtie$ is associative, up to the state names, because the composition of labels $\otimes$ is associative. From $P_1 \cap P_3 = \emptyset$, $\ell_1 \in \mathsf{L}[P_1]$, and from the locality property in Definition 4 we conclude that $\forall q \in Q_3 \cdot \ell_1^{(P_3)} \notin \mathcal{C}_3(q)$. Therefore, for any state $q_3 \in Q_3$ and for a state $q_2 \in Q_2$ such that $\ell_1^{(P_2)} \notin \mathcal{C}_2(q_2)$, we have that $\ell_1^{(P_2)} \notin \mathcal{C}_2(q_2) \cup \mathcal{C}_3(q_3)$. We conclude that $\ell_1^{(P_2 \cup P_3)} \notin \mathcal{C}'$, where $\mathcal{C}'$ is the concurrency predicate of $b_2 \bowtie b_3$, hence a local step of b. □

If the locality property holds for each behavioural automata b_i in a composed system $b = b_1 \bowtie \cdots \bowtie b_n$, then, using Proposition 1, we can infer wether atomic steps from b_i are local steps of b based only on the concurrency predicates of its *neighbour automata*, i.e., the automata that share ports with b_i.

2.5 Concrete behavioural automata

A behavioural automaton is an abstraction of concrete coordination models that focuses on aspects relevant to the execution of the coordination model. As we will argue, Reo and Linda can be cast in our framework of behavioural automata. Therefore, both Reo and Linda coordination models can be seen as specific instances of the stepwise model described above. For a concrete coordination model to fit into the stepwise model, we need to define: (1) labels in the concrete model; (2) the encoding α of labels into atomic steps; (3) composition of labels; and (4) concurrency predicates.

We start by encoding the constraint automata semantics of Reo as behavioural automata. Later, because of its relevance in the coordination community as one of the first coordination languages, we also encode Linda as a behavioural automaton. Other coordination models have also been encoded as behavioural automata in Proença's Ph.D. thesis [18].

3 Encoding Reo

Reo [1, 2] is presented as a channel-based coordination language wherein component connectors are compositionally built out of an open set of *primitive connectors*, also called primitives. Channels are primitives with two ends. Existing tools for Reo include an editor, an animation generator, model checkers, editors of Reo-specific automata, QoS modelling and analysis tools, and a code generator [5, 16].

The behaviour of each primitive depends upon its current state.[3] The semantics of a connector is described as a collection of possible steps for each state, and we call the change of state of the connector triggered by one of these steps a *round*. At each round some of the ends of a connector are synchronised, i.e., only certain combinations of synchronous dataflow through its ends are possible. Dataflow on a primitive's end occurs when a single datum is passed through that end. Within any round dataflow may occur on some number of ends. Communication with a primitive connector occurs through its ports, called *ends*. Primitives consume data through their *source ends*, and produce data through their *sink ends*. Connectors are formed by plugging the ends of primitives together in a one-to-one fashion to form *nodes*. A node is a logical place consisting of a sink end, a source end, or both a sink and a source end.[4]

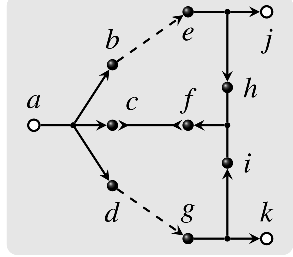
We now give an informal description of some of the most commonly used Reo primitives. Note that, for all of these primitives, no dataflow is one of the behavioural possibilities. The Sync channel

[3] Note that most Reo primitives presented here have a single state.

[4] Generalised nodes with multiple sink and source ends can be defined by combining binary mergers and replicators [6, 11].

(⟶) sends data synchronously from its source to its sink end. The LossySync channel (⇢) differs from the Sync channel only because it can non-deterministically lose data received from its source port. The SyncDrain (↣) has two source ends, and requires both ends to have dataflow synchronously, or no dataflow is possible. The FIFO_1 channel (⟶□⟶) has two possible states: empty or full. When empty, it can receive a data item from its source end, changing its state to full. When full, it can only send the data item received previously, changing its state back to empty. Finally, a replicator ( ) replicates data synchronously to all of its sink ends, while a merger ( ) copies data atomically from exactly one of its sink ends to its source end.

Example 3 *The connector on the right is an exclusive router built by composing two LossySync channels (b-e and d-g), one SyncDrain (c-f), one Merger (h-i-f), and three Replicators (a-b-c-d, e-j-h and g-i-k). The constraints of these primitives can be combined to give the following two behavioural possibilities (plus the no-flow-everywhere possibility):*



- *ends* $\{a,b,c,d,e,f,h,j\}$ *synchronise and a data item flows from a to j,*
- *ends* $\{a,b,c,d,f,g,i,k\}$ *synchronise and a data item flows from a to k.*

The merger makes a non-deterministic choice whenever both behaviours are possible. Data can never flow from a to both j and k, as this is excluded by the behavioural constraints of the Merger h-i-f.

3.1 Constraint automata

We briefly describe constraint automata [6]. Constraint automata use a finite set of port names $\mathcal{N} = \{x_1,\ldots,x_n\}$, where x_i is the i-th port of a connector. When clear from the context, we write xyz instead of $\{x,y,z\}$ to enhance readability. We write $\widehat{x_i}$ to represent the variable that holds the data value flowing through the port x_i, and use $\widehat{\mathcal{N}}$ to denote the set of data variables $\{\widehat{x}_1,\ldots,\widehat{x}_n\}$, for each $x_i \in \mathcal{N}$. We define DC_X for each $X \subseteq \mathcal{N}$ to be a set of data constraints over the variables in $\widehat{X}$, where the underlying data domain is a finite set $\mathbb{D}$. Data constraints in $DC_{\mathcal{N}}$ can be viewed as a symbolic representation of sets of data-assignments, and are generated by the following grammar:

$$g \ ::= \ \mathtt{tt} \ \big| \ \widehat{x} = d \ \big| \ g_1 \vee g_2 \ \big| \ \neg g$$

where $x \in \mathcal{N}$ and $d \in \mathbb{D}$. The other logical connectives can be encoded as usual. We use the notation $\widehat{a} = \widehat{b}$ as a shorthand for the constraint $(\widehat{a} = d_1 \wedge \widehat{b} = d_1) \vee \ldots \vee (\widehat{a} = d_n \wedge \widehat{b} = d_n)$, with $\mathbb{D} = \{d_1,\ldots,d_n\}$.

Definition 5 (Constraint Automaton [6]) *A* constraint automaton *(over the finite data domain* $\mathbb{D}$*) is a tuple* $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow, Q_0 \rangle$*, where* Q *is a set of states,* $\mathcal{N}$ *is a finite set of port names,* $\rightarrow$ *is a subset of* $Q \times \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}} \times Q$*, called the transition relation of* $\mathcal{A}$*, and* $Q_0 \subseteq Q$ *is the set of initial states.*

We write $q \xrightarrow{X|g} p$ instead of $(q,X,g,p) \in \rightarrow$. For every transition $q \xrightarrow{X|g} p$, we require that g, the guard, is a DC_X-constraint. For every state $q \in Q$, there is a transition $q \xrightarrow{\emptyset|\mathtt{tt}} q$.

We define $\mathsf{CAS} \subseteq \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}}$ to be the set of solutions for all possible labels of the transitions of constraint automata. That is, $X|g \in \mathsf{CAS}$ if $X = \{x_1,\ldots,x_n\}$, $g = \bigwedge \widehat{x_i} = v_i$, where $v_i \in \mathbb{D}$, and there is a transition $q \xrightarrow{X|g'} q'$ such that g satisfies g'. We call each $s \in \mathsf{CAS}$ a constraint automaton step. Firing a transition $q \xrightarrow{X|g} p$ is interpreted as having dataflow at all the ports in X, while excluding flow at ports in $\mathcal{N} \setminus X$, when the automaton is in the state q. The data flowing through the ports X must satisfy the constraint g, and the automaton evolves to the state p. Figure 5 exemplifies the constraint automata for three Reo channels. We do not define here the composition of constraint automata, but encode labels of constraint automata as labels of behavioural automata, whose composition has been defined in §2.2.

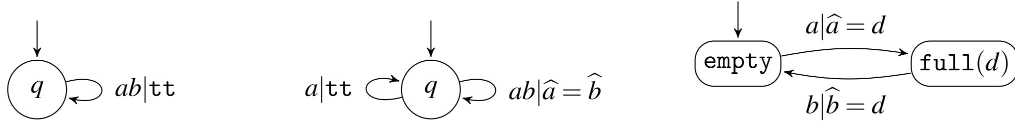


Figure 5: From left to right, constraint automata for the SyncDrain, LossySync and FIFO$_1$ channels.

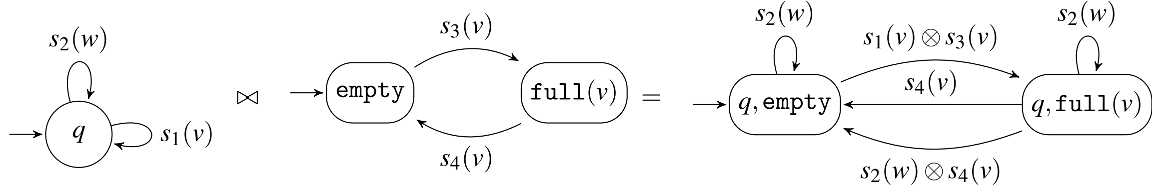


Figure 6: Composition of $[\![\mathcal{A}_L]\!]_{\mathsf{CA}}$ and $[\![\mathcal{A}_F]\!]_{\mathsf{CA}}$, for any $v,w \in \mathbb{D}$.

3.2 Constraint automata as behavioural automata

The CA model assumes a finite data domain $\mathbb{D}$, and that data constraints such as $\mathtt{tt}$, $\widehat{a} \neq d$, or $\widehat{a} = \widehat{b}$ stand for simpler data constraints that use $\widehat{a} = d$ and the operators $\wedge$ and $\vee$.

The encoding of the constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow_{\mathsf{CA}}, Q_0 \rangle$ is the behavioural automaton

$$[\![\mathcal{A}]\!]_{\mathsf{CA}} = \langle Q, \mathsf{L}[\mathcal{N}], \rightarrow_{\mathsf{BA}}, \mathcal{C} \rangle$$

with $\mathsf{L}[\mathcal{N}]$, $\rightarrow_{\mathsf{BA}}$, $\mathcal{C}$, and the composition of labels defined as follows:

- $\mathsf{L} = \mathsf{CAS}$, and α is defined as: $\alpha(X | \bigwedge_{i=1}^{n} \widehat{x_i} = d_i) \; = \; \langle \mathcal{N}, X, \emptyset, X, \{x_i \mapsto d_i\}_{i=1}^{n} \rangle$.
- We have $q \xrightarrow{X|g}_{\mathsf{BA}} q'$ for $X|g \in \mathsf{L}[\mathcal{N}]$ if $q \xrightarrow{X|g'}_{\mathsf{CA}} q'$ and g satisfies g'.
- Let $cas_i = X_i | g_i$ be a solution for a label in a constraint automaton with ports $\mathcal{N}_i$, for $i \in 1..2$. Then

  $$cas_1 \otimes cas_2 = \begin{cases} (X_1 \cup X_2) | (g_1 \wedge g_2) & \text{if } X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1 \; \wedge \; g_1 \frown g_2 \\ \bot & \text{otherwise} \end{cases}$$

  where $g_1 \frown g_2$ if for any port $x \in X_1 \cap X_2$ and for any $d \in \mathbb{D}$, $x = d$ satisfies g_1 iff $x = d$ satisfies g_2.
- $\mathcal{C}(q) = cp(\mathcal{N})$ for every $q \in Q$. Recall that $cp(\mathcal{N}) = \{\ell \mid \alpha(\ell) = \langle P, F, IP, OP, data \rangle, P_0 \cap F \neq \emptyset\}$.

Example 4 *Let $\mathcal{A}_L = \langle Q_L, \mathcal{N}_L, \rightarrow_1, Q_1 \rangle$ and $\mathcal{A}_F = \langle Q_F, \mathcal{N}_F, \rightarrow_2, Q_2 \rangle$ be the constraint automata of the LossySync and the FIFO$_1$ channels, depicted in Figure 5. The encoding of $\mathcal{A}_L$ into behavioural automata is $\langle Q_L, \mathsf{L}[\mathcal{N}_L], \rightarrow_L, \mathcal{C}_L \rangle$, depicted in the left hand side of Figure 6, where:*
$Q_L = \{q\}$, $\mathcal{N}_L = \{a,b\}$, $\mathcal{C}_L(q) = cp(\mathcal{N}_L)$ for $q \in Q_L$, $s_1(v) = ab|(\widehat{a} = v \wedge \widehat{b} = v)$, $s_2(v) = a|(\widehat{a} = v)$, and $\rightarrow_L = \{\langle q, s_1(v), q \rangle \mid v \in \mathbb{D}\} \cup \{\langle q, s_2(v), q \rangle \mid v \in \mathbb{D}\}$.

Similarly, the encoding of $\mathcal{A}_F$ into behavioural automata is $\langle Q_F, \mathsf{L}[\mathcal{N}_F], \rightarrow_F, \mathcal{C}_F \rangle$, also depicted in Figure 6, where:
$Q_F = \{\mathtt{empty}\} \cup \{\mathtt{full}(v) \mid v \in \mathbb{D}\}$, $\mathcal{C}_F(q) = cp(\mathcal{N}_F)$ for $q \in Q_F$, $\mathcal{N}_F = \{b,c\}$, $s_3(v) = b|(\widehat{b} = v)$, $s_4(v) = c|(\widehat{c} = v)$, and $\rightarrow_F = \{\langle \mathtt{empty}, s_3(v), \mathtt{full}(v) \rangle \mid v \in \mathbb{D}\} \cup \{\langle \mathtt{full}(v), s_4(v), \mathtt{empty} \rangle \mid v \in \mathbb{D}\}$.

The composed automaton $[\![\mathcal{A}_L]\!]_{CA} \bowtie [\![\mathcal{A}_F]\!]_{CA}$ is depicted in the right hand side of Figure 6, where $s_1(v) \otimes s_3(v) \; = \; ab|(\widehat{a} = v \wedge \widehat{b} = v)$ and $s_2(w) \otimes s_4(v) \; = \; ac|(\widehat{a} = w \wedge \widehat{c} = v)$.

The composed automata presented in Example 4, which differs from the lossy-FIFO, is equivalent to the product of the two associated constraint automata [6], with respect to the atomic steps of the labels of the automata. We expect this equivalence to hold in general, but we do not give a formal proof here.

4 Encoding Linda

Linda, introduced by Gelernter [15], is seen by many as the first coordination language. We describe it using Linda-calculus [10], and show how it can be modelled using behavioural automata. Linda is based on the *generative communication* paradigm, which describes how different processes in a distributed environment exchange data. In Linda, data objects are referred to as *tuples*, and multiple processes can communicate using a *shared tuple-space*, where they can write or read tuples.

Communication between processes and the tuple-space is done by actions executed by processes over the tuple-space. In general, these actions can occur only atomically, that is, the shared tuple-space can accept and execute an action from only one of the processes at a time. There are four possible actions, $\mathbf{out}(t)$, $\mathbf{in}(s)$, $\mathbf{rd}(s)$, and $\mathbf{eval}(P)$. The actions $\mathbf{out}(t)$ and $\mathbf{in}(s)$ write and take values to and from the shared tuple-space, respectively. The action $\mathbf{rd}(s)$ is similar to $\mathbf{in}(s)$, except that the tuple t is not removed from the tuple-space. Finally, $\mathbf{eval}(P)$ denotes the creation of a new process P that will run in parallel. We do not address $\mathbf{eval}(P)$ here because it is regarded as a reconfiguration of the system.

4.1 Linda-Calculus

We use the Linda-Calculus model, described by Goubault [12], to give a formal description of Linda, studied also by Ciancarini *et al.* [10] and others. The Linda-Calculus abstracts away from the local behaviour of processes, and focuses on the communication primitives between a *store* and a set of *processes*. Processes P are generated by the following grammar.

$$P ::= Act.P \;\big|\; X \;\big|\; \mathbf{rec}X.P \;\big|\; P \,\square\, P \;\big|\; \mathbf{end} \tag{5}$$

$$Act ::= \mathbf{out}(t) \;\big|\; \mathbf{in}(s) \;\big|\; \mathbf{rd}(s) \tag{6}$$

We denote the set of all Linda terms as Linda. The first case $Act.P$ represents the execution of a Linda action. The productions X and $\mathbf{rec}X.P$ are used to model recursive processes, where X ranges over a set of variables, and $P \,\square\, P$ is used to model local non-deterministic choice. We assume that processes do not have free variables, i.e., every X is bound by a corresponding $\mathbf{rec}X$. Finally $\mathbf{end}$ represents termination.

We model a Linda store as a multi-set of tuples from a global set *Tuple*. Each tuple consists of a sequence of parameters, which can be either a data value v from a domain $\mathbb{D}$ (an actual parameter), or a variable X (a formal parameter). We use the $\oplus$ operator to denote multi-set construction and multi-set union. For example, we write $M = t \oplus t = \{|t,t|\}$ and $M \oplus M = \{|t,t,t,t|\}$, where t is a tuple and $\{|s,t|\}$ denotes a multi-set with the elements s and t.

A *tuple-space term* M is a multi-set of processes and tuples, generated by the grammar $M ::= P \mid t \mid M \oplus M$. We adopt the approach of Goubault and provide a set of inference rules that give the operational semantics of Linda-Calculus. A relation $match \subseteq Tuple \times Tuple$ represents the matching of two tuples. $(s,t) \in match$ if t has only $\mathbb{D}$ values, and there is a substitution γ whose domain is the set of free variables of s, such that $t = s[\gamma]$. $u[\gamma]$ denotes the tuple or process u after replacing its free variables according to γ. We also write $\gamma = P/x$ to denote the substitution of x by the process P, and say t γ-matches s when t matches s and $t = s[\gamma]$.

Definition 6 (Semantics of Linda) *The semantics of Linda is defined by the inference rules below.*

$$\frac{M \oplus P[\mathbf{rec}X.P/X] \longrightarrow M \oplus P'}{M \oplus \mathbf{rec}X.P \longrightarrow M \oplus P'} \quad \text{(rec)}$$

$$M \oplus P \,\square\, P' \longrightarrow M \oplus P \quad \text{(left)}$$

$$M \oplus P \,\square\, P' \longrightarrow M \oplus P' \quad \text{(right)}$$

$$M \oplus \mathbf{out}(t).P \longrightarrow M \oplus P \oplus t \quad \text{(out)}$$

$$M \oplus \mathbf{rd}(s).P \oplus t \longrightarrow M \oplus P[\gamma] \oplus t \quad \textit{if } t \ \gamma\textit{-matches } s \quad \text{(rd)}$$

$$M \oplus \mathbf{in}(s).P \oplus t \longrightarrow M \oplus P[\gamma] \quad \textit{if } t \ \gamma\textit{-matches } s \quad \text{(in)}$$

$$M \oplus \mathbf{end} \longrightarrow M \quad \text{(end)}$$

Example 5 *The following sequence of transitions illustrates the sending of data between two processes. The labels on the arrows contain the names of the rules applied in each transition of Linda-Calculus. We use the notation $P(x)$ as syntactic sugar to denote a process P where the variable x occurs freely.*

$$\begin{array}{llll} & \mathbf{rd}(42,x).P(x) \oplus \mathbf{out}(42,43).\mathbf{end} \oplus \mathbf{in}(42,x).P'(x) & & \\ \xrightarrow{(out)} & \mathbf{rd}(42,x).P(x) \oplus \mathbf{end} \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle & \xrightarrow{(end)} & \mathbf{rd}(42,x).P(x) \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle \\ \xrightarrow{(rd)} & P(43) \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle & \xrightarrow{(in)} & P(43) \oplus P'(43) \end{array}$$

4.2 Linda-calculus as behavioural automata

We define an encoding function $[\![\cdot]\!]_{\mathsf{Linda}} : \mathsf{Linda} \to \mathsf{BA}$, from Linda tuple-space terms to behavioural automata. Furthermore, we define the composition of atomic steps that preserve this semantics. We encode each Linda process P as a behavioural automaton, and we create a special behavioural automaton that describes the multi-set of available tuples.

Let $\overline{Act} = \{\overline{a} \mid a \in Act\}$ and $\tau Act = \{\tau_a \mid a \in Act\}$. A port $\overline{a}$ is regarded as a dual port of a, and flow of data on a port τ_a represents the flow on the ports a and $\overline{a}$ simultaneously. The intuition is that the encoding of processes yields behavioural automata whose ports are actions in Act; the encoding of tuples yield behavioural automata whose ports are *dual* actions in $\overline{Act}$; and the composition forces actions and dual actions to synchronise, i.e., to occur simultaneously. We define the global set of ports to be $\mathbb{P} = Act \cup \overline{Act} \cup \tau Act$, and define $\overline{\overline{a}} = a$.

Let $M = P_1 \oplus \cdots \oplus P_n \oplus T$ be a tuple-space term. In turn, let $T = t_1 \oplus \cdots \oplus t_m$ and $m \geq 0$. We define the encoding of M into a behavioural automaton as follows.

$$[\![M]\!]_{\mathsf{Linda}} = [\![P_1]\!]_{\mathsf{Linda}} \bowtie \cdots \bowtie [\![P_n]\!]_{\mathsf{Linda}} \bowtie [\![T]\!]_{\mathsf{Linda}}$$

Hence, encoding M boils down to encoding Linda processes P_i and the Linda tuple-space T into different behavioural automaton. In both encodings of components and Linda tuple-spaces we define labels L as ports, that is, $\mathsf{L} = \mathbb{P} = Act \cup \overline{Act} \cup \tau Act$, and its encoding as atomic steps by the function α defined below.

$$\alpha(a) = \begin{cases} \langle \mathbb{P}, \{a, \tau_{act}\}, \emptyset, \emptyset, \emptyset \rangle & \text{if } a \in Act \cup \overline{Act}, \{act\} = \{a, \overline{a}\} \cap Act \\ \langle \mathbb{P}, \{a\}, \emptyset, \emptyset, \emptyset \rangle & \text{if } a \in \tau Act \end{cases}$$

The composition of two labels $a_1, a_2 \in \mathsf{L}$ is defined as follows.

$$a_1 \otimes a_2 = \begin{cases} \tau_{act} & \text{if } a_1 \notin \tau Act \wedge a_2 \notin \tau Act \wedge a_1 = \overline{a}_2 \\ \bot & \text{otherwise,} \end{cases}$$

where $\{act\} = \{a_1, a_2\} \cap Act$. The tuple-space is used to enforce every action a performed by a process to synchronise with the corresponding action $\overline{a}$ in the tuple-space encoded as a behavioural automaton. The definition of $\otimes$ replaces every pair of ports with dataflow a and $\overline{a}$ by a new port with dataflow in τ_{act}.

We encode a Linda process P as $[\![P]\!]_{\mathsf{Linda}} = \langle Q_P, \mathsf{L}, \to_P, \mathcal{C} \rangle$, with components as defined below.

- The set of states Q_P is given by $Q_P = reach(P)$, where

$$\begin{array}{rcl} reach(\mathbf{out}(t).P) & = & \{\mathbf{out}(t).P\} \cup reach(P) \\ reach(\mathbf{rd}(s).P) & = & \{\mathbf{rd}(t).P\} \cup (\bigcup \{reach(P[\gamma]) \mid s\ \gamma\text{-matches } t\}) \\ reach(\mathbf{in}(s).P) & = & \{\mathbf{in}(t).P\} \cup (\bigcup \{reach(P[\gamma]) \mid s\ \gamma\text{-matches } t\}) \\ reach(P \Box P') & = & \{P \Box P'\} \cup reach(P) \cup reach(P') \\ reach(\mathbf{end}) & = & \{\mathbf{end}\} \end{array}$$

- The transition relation $\rightarrow_P$ is given by the following conditions.

$$\begin{array}{rlll} \mathbf{out}(t).P' & \xrightarrow{\mathbf{out}(t)} P' & \textit{if} & t \in \textit{Tuple} \\ \mathbf{rd}(s).P' & \xrightarrow{\mathbf{rd}(t)} P'[\gamma] & \textit{if} & s\ \gamma\textit{-matches}\ t \\ \mathbf{in}(s).P' & \xrightarrow{\mathbf{in}(t)} P'[\gamma] & \textit{if} & s\ \gamma\textit{-matches}\ t \end{array} \qquad \begin{array}{llll} P_1 \,[\!]\, P_2 \xrightarrow{s} P_1' & \textit{if} & P_1 \xrightarrow{s} P_1' \\ P_1 \,[\!]\, P_2 \xrightarrow{s} P_2' & \textit{if} & P_2 \xrightarrow{s} P_2' \end{array}$$

- $\mathcal{C}(q) = Act \cup \overline{Act}$ for every state q.

We now encode a Linda tuple-space T as $[\![T]\!]_{\mathsf{Linda}} = \langle Q_T, \mathsf{L}, \rightarrow_T, \mathcal{C}\rangle$ with components as defined below.

- $Q_T = \mathbf{2}^{\mathcal{M}(Tuple)}$, where $\mathcal{M}(X)$ is a multi-set over the set X.
- The transition relation $\rightarrow_T$ is given by the following conditions:
$M \xrightarrow{\overline{\mathbf{out}(t)}} M \oplus t$ *if* $t \in Tuple$, $\quad t \oplus M \xrightarrow{\overline{\mathbf{rd}(s)}} t \oplus M$ *if* s *matches* t, and $\quad t \oplus M \xrightarrow{\overline{\mathbf{in}(s)}} M$ *if* s *matches* t.
- $\mathcal{C}(q) = Act \cup \overline{Act}$ for every state q, as in the encoding of Linda processes.

Note that the input and output ports of the atomic steps obtained with α, introduced in §2.1, are always the empty set, that is, the data value flowing through the ports is not relevant, since the name of the port uniquely identifies the data. Alternative approaches to implement the encoding into behavioural automata that use the data values are also possible, but less transparent.

Example 6 *Recall the example presented in the end of §4.1 of a sequence of transitions of a tuple-space term in Linda-Calculus. We present below a simplified version of this example.*

$$\mathbf{rd}(42,x).P(x) \oplus \mathbf{out}(42,43).P' \xrightarrow{(out)} \mathbf{rd}(42,x).P(x) \oplus P' \oplus \langle 42,43\rangle \xrightarrow{(rd)} P(43) \oplus P' \oplus \langle 42,43\rangle$$

The corresponding transitions in the encoded behavioural automaton are presented below.

$$\begin{array}{l} [\![\mathbf{rd}(42,x).P(x)]\!]_{\mathsf{Linda}} \bowtie [\![\mathbf{out}(42,43).P']\!]_{\mathsf{Linda}} \bowtie [\![\emptyset]\!]_{\mathsf{Linda}} \xrightarrow{\tau_{\mathbf{out}(42,43)}} \\ [\![\mathbf{rd}(42,x).P(x)]\!]_{\mathsf{Linda}} \bowtie [\![P']\!] \bowtie [\![\langle 42,43\rangle]\!] \xrightarrow{\tau_{\mathbf{rd}(42,43)}} [\![P(43)]\!]_{\mathsf{Linda}} \bowtie [\![P']\!] \bowtie [\![\langle 42,43\rangle]\!] \end{array}$$

Observe that we assume an initial empty tuple-space, which is encoded as $[\![\emptyset]\!]_{\mathsf{Linda}}$. A more careful analysis shows a one-to-one correspondence between the traces of the Linda-calculus term and the traces of the behavioural automaton, which we do not elaborate in this paper.

5 Exploiting concurrency predicates

We introduced a unified model for synchronous coordination that explicitly mentions concurrency predicates, which indicate which actions require synchronisation. We now exploit more complex definitions of concurrent predicates for Reo and Linda than in our previous examples, and briefly describe a practical application of behavioural automata in a distributed framework.

5.1 Complex concurrency predicates

In our examples concurrency predicates of Reo hold when some shared ports from a composed automaton have dataflow (Equation (1)), and concurrency predicates of Linda allow only a special set of actions τAct to run concurrently. We now present other concurrency predicates that capture notions such as context dependency and priority.

Reo Other semantic models for Reo, such as connector colouring [11] and Reo automata [8], capture the notion of *context dependency*, a feature missing in constraint automata. By modelling context dependency we avoid the undesired behaviour of the composed connector in Figure 6 where data is lost when the FIFO_1 buffer is empty, represented by the label $s_2(w)$.

To avoid data from being lost, we replace the LossySync channel by a context dependent LossySync channel, which is built based on the LossySync channel by replacing the label $s_2(w)$ by a label $s_2^b(w)$. This new label has the same atomic step, i.e., $\alpha(s_2(w)) = \alpha(s_2^b(w))$, but can be executed in parallel only if its neighbours require the port b to have no dataflow. This condition is enforced by adapting the definition of concurrency predicates to check wether a given set of ports Y requires synchronisation.

$$cp_{ctx}(P_0,Y) = \left\{s^X \mid s^X \in cp(P_0) \;\;\vee\;\; X \cap Y \neq \emptyset\right\} \tag{7}$$

In our example, we avoid the losing of data by defining $\mathcal{C}(q) = cp_{ctx}(ab,\emptyset)$, $\mathcal{C}(\texttt{empty}) = cp_{ctx}(bc,b)$, and $\mathcal{C}(\texttt{full}(v)) = cp_{ctx}(ab,c)$. The label $s_2^b(w)$ is in $\mathcal{C}(\texttt{empty})$ but not in $\mathcal{C}(\texttt{full})$, i.e., $s_2^b(w)$ can be performed independently of the FIFO_1 channel only when the FIFO_1 is full. Other important details, such as the composition of labels of the form s^X, are not presented in this paper. A more precise and complete formulation can be found in Proença's Ph.D. thesis (Sections 3.6.2 and 4.4.2 of [18]).

Linda Consider now that Linda processes have a total order $\preceq$, representing a ranking among processes. When two processes can interact simultaneously with the shared tuple-space, only the higher rank should be chosen. We present only a sketch of this approach due to space limitation.

We start by tagging labels ℓ of the Linda behavioural automata with the process that executes it. For example, a label ℓ of an automaton of a process p is renamed to ℓ^p. Labels of the shared tuple-space are not changed. The composition of labels must be such that $\ell^p \otimes \overline{\ell} = \tau_\ell^p$. It is then enough to change the concurrency predicates of the automata of each process p in state q to $\mathcal{C}(q) = Act \cup \overline{Act} \cup \left\{\tau_\ell^x \mid \tau_\ell \in \tau Act \;\wedge\; x \preceq p \;\wedge\; q \neq \textbf{end}\right\}$ and leave the concurrency predicate of the automaton of the shared tuple-space unchanged. Hence, a transition cannot be performed in parallel if it is in Act or $\overline{Act}$, or if it is a τ action from a process with lower priority and the current process is not yet stopped.

5.2 Increased scalability via decoupled execution

We use the behavioural automata model in a distributed framework, Dreams, where several independent threads run concurrently [18]. Each thread has its own behavioural automaton, and communicates only with those threads whose behavioural automata share ports with its own automata. The details regarding this tool are out of the scope this paper, but we explain how it benefits from using behavioural automata.

The diagram in Figure 7 depicts the configuration of a system in Dreams, where each cloud represents an independent thread of execution, and edges represent communication links between threads whose automata share ports. The direction of each edge only illustrates the expected direction of dataflow. For efficiency reasons, and to allow a lightweight reconfiguration, Dreams does not create the complete behavioural automaton of a connector. Instead, it collects only the behaviour of the current round.

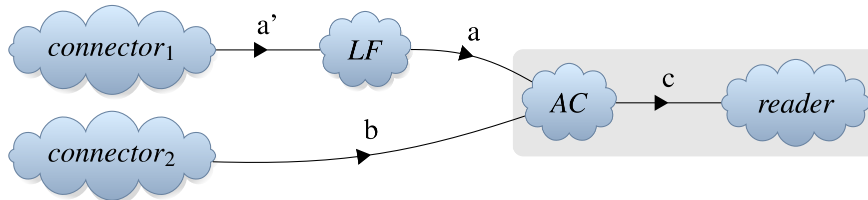


Figure 7: Configuration of a system in Dreams.

Knowing that only the labels of the automata relevant for the current round are composed, and assuming that the locality property introduced in Definition 4 holds, we can perform local steps that, as the name suggests, involve only a subpart of the system. Recall the example of the lossy alternator, presented in §2.3. The diagram in Figure 7 uses the same example, in a context where two arbitrary large connectors $connector_1$ and $connector_2$ are attached to the source of the lossy alternator, and a *reader* component is attached to the sink of the lossy alternator. Consider that the *reader* can always receive any data value, that is, its behavioural automaton has a single state, and a transition labelled by $r(v)$ for every data value v, such that $\alpha(r(v)) = \langle c, c, c, \emptyset, \{c \mapsto v\}\rangle$.

Observe that we do not use explicitly the composed connector $LF \bowtie AC$, but LF and AC as independent entities instead, since the Dreams framework can postpone the composition of their labels to runtime. Consider that the AC automaton is in state $q_1(v)$, hence it can perform a step $s_2(v)$, writing a value v to the port c. In this example AC is connected via the ports a, b, and c. The label $s_2(v)$ does not have dataflow on a nor on b, and the reader can perform a label $r(v)$ because $s_2(v) \otimes r(v) \neq \perp$. Using the concurrency predicate in Equation (1), we conclude that $s_2(v) \otimes r(v)$ is in the concurrency predicates of LF and $connector_2$. Furthermore, from the locality property we conclude that all other connectors not attached to AC also allow $s_2(v) \otimes r(v)$ to be executed concurrently. Hence, Dreams can chose to perform this step by analysing only the behaviour of AC and *reader*, depicted by a grey box.

The instantiations of Linda and Reo yield a similar result. The shared tuple-space can communicate with a single process at a time, without synchronising with every other process. Reo can, for example, send data from a full FIFO_1 independently of the behaviour of the connector attached to its sink port. The benchmarks performed for the Dreams framework [18] show optimistic results regarding the use of local steps in synchronous coordination.

6 Conclusion

We introduce behavioural automata to model coordination systems. The three main concepts that underlie behavioural automata are *atomicity*, *composability*, and *dataflow*. We allow a sequence of actions that cannot be interleaved with interfering instructions (atomicity), we construct more complex systems out of building blocks that can be analysed independently (composability), and we represent the data values that are exchanged between components (dataflow).

Behavioural automata unify existing dataflow-oriented models with synchronous constructs by leaving open the definitions of composition of labels and of concurrency predicates. The focus of behavioural automata is on concurrent systems, and on avoiding synchronisation of actions whenever it is unnecessary. By capturing a multitude of coordination models, we allow any of these models to be included in implementations based on behavioural automata, such as the Dreams framework.

As future work, we expect to formally show the correctness of the encodings of Reo and Linda. We

would also like to discover which properties can be shown for behavioural automata that are directly reflected on encoded models. A more practical track of our work involves the development of tools. Further development of Dreams to make it ready for use by a broader community is in our agenda.

References

[1] Farhad Arbab (2004): *Reo: a channel-based coordination model for component composition*. Mathematical Structures in Computer Science 14(3), pp. 329–366.

[2] Farhad Arbab (2005): *Abstract Behavior Types: a foundation model for components and their composition*. Science of Computer Programming 55, pp. 3–52.

[3] Farhad Arbab (2006): *Composition of Interacting Computations*, chapter 12, pp. 277–321. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[4] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese & Ugo Montanari (2009): *Tiles for Reo*. In: Recent Trends in Algebraic Development Techniques, LNCS 5486, Springer, pp. 37–55.

[5] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon & José Proença (2008): *Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools*. In: Proceedings of FACS, SCP.

[6] Christel Baier, Marjan Sirjani, Farhad Arbab & Jan J. M. M. Rutten (2006): *Modeling component connectors in Reo by constraint automata*. Science of Computer Programming 61(2), pp. 75–113.

[7] Gérard Berry (2000): *The foundations of Esterel*. In Gordon D. Plotkin, Colin Stirling & Mads Tofte, editors: Proof, Language, and Interaction, The MIT Press, pp. 425–454.

[8] Marcello M. Bonsangue, Dave Clarke & Alexandra Silva (2009): *Automata for Context-Dependent Connectors*. In: COORDINATION, LNCS 5521, Springer, pp. 184–203.

[9] D. M. Chapiro (1984): *Globally-Asynchronous Locally-Synchronous Systems*. Ph.D. thesis, Standford University.

[10] P. Ciancarini, K.K. Jensen & D. Yankelevich (1995): *On the Operational Semantics of a Coordination Language*. In: Object-Based Models and Languages for Concurrent Systems, LNCS 924, pp. 77–106.

[11] Dave Clarke, David Costa & Farhad Arbab (2007): *Connector colouring I: Synchronisation and context dependency*. Science of Computer Programming 66(3), pp. 205–225.

[12] Régis Cridlig & Eric Goubault (1993): *Semantics and Analysis of Linda-Based Languages*. In Patrick Cousot, Moreno Falaschi, Gilberto Filé & Antoine Rauzy, editors: WSA, LNCS 724, Springer, pp. 72–86.

[13] Frederic Doucet, Massimiliano Menarini, Ingolf H. Krüger, Rajesh K. Gupta & Jean-Pierre Talpin (2006): *A Verification Approach for GALS Integration of Synchronous Components*. ENTCS 146(2), pp. 105–131.

[14] Fabio Gadducci & Ugo Montanari (2000): *The tile model*, pp. 133–166. MIT Press, Cambridge, MA, USA.

[15] David Gelernter (1985): *Generative communication in Linda*. ACM Transactions on Programming Languages and Systems 7(1), pp. 80–112.

[16] Christian Krause (2011): *Reconfigurable component connectors*. Ph.D. thesis, Leiden University.

[17] Robin Milner (1983): *Calculi for Synchrony and Asynchrony*. Theor. Comput. Sci. 25, pp. 267–310.

[18] José Proença (2011): *Synchronous Coordination of Distributed Components*. Ph.D. thesis, Leiden University.