Overview of Thirty Semantic Formalisms for Reo

Sung-Shik T.Q. JONGMANS\textsuperscript{1}, Farhad ARBAB\textsuperscript{2}

Abstract

Over the past decades, coordination languages have emerged for the specification and implementation of interaction protocols for communicating software components. This class of languages includes Reo, a platform for compositional construction of connectors. In recent years, many formalisms for describing the behavior of Reo connectors have emerged. In this paper, we give an overview of all these classes of semantic models. Furthermore, we investigate the expressiveness of two more prominent classes, constraint automata and coloring models, in detail.

Keywords: concurrency, coordination, Reo, semantics, constraint automata, connector coloring

1 Introduction

Over the past decades, coordination languages have emerged for the specification and implementation of interaction protocols for communicating software components. This class of languages includes Reo\textsuperscript{5}, a platform for compositional construction of connectors. Connectors in Reo serve as communication mediums through which components can interact with each other. Essentially, Reo connectors impose constraints on the order in which components can exchange data items with each other. Although ostensibly simple, Reo connectors can describe complex protocols (e.g., a solution to the Dining Philosophers problem\textsuperscript{6}). Development tools for Reo exist as plug-ins for the Eclipse IDE, called the Extensible Coordination Tools (ECT),\textsuperscript{3,4} including tools for simulation, animation, and verification of Reo connectors.

\textsuperscript{1}CWI, Amsterdam, the Netherlands. Email: jongmans@cwi.nl
\textsuperscript{2}CWI, Amsterdam, the Netherlands; LIACS, Leiden, the Netherlands
\textsuperscript{3}\url{http://reo.project.cwi.nl}
\textsuperscript{4}Formerly known as the Eclipse Coordination Tools.
In recent years, many semantic formalisms for describing the behavior of Reo connectors have emerged, including coalgebraic models, operational models, and models based on graph-coloring. Table 1 shows a list of the classes of semantic models that currently (late 2011) exist for modeling Reo connectors. (We discuss the entries in this list in more detail in Section 3.) Although each of the classes in Table 1 serves its own purpose, we identify two burdens that their large quantity inflicts.

**Issue 1** (Tracking them). *Because there exist so many different semantic formalisms by now, keeping track of all of them becomes nontrivial and requires a significant amount of effort. As a result, in more than one publication, authors forget to mention relevant related work at the appropriate places in their own contributions. Extrapolated to the extreme, this can cause papers to become “forgotten” and scientist to redo the work.*

**Issue 2** (Relating them). *Questions about how the various classes of models relate to each other in terms of their expressiveness naturally arise. These questions, moreover, require answering for two reasons. First, from a theoretical point of view, such answers provide us with better and possibly new insights into the fundamentals of Reo. Second, from a more practical perspective, such answers broaden the applicability of tools—both existing and*
future—that assist developers in designing their connectors.

In this paper, we provide a snapshot that offers a solution to Issue 1 and take a step towards answering the questions mentioned in Issue 2.

Contributions We attribute two main contributions to this paper. In its first half, to resolve Issue 1, we provide a comprehensive overview of all existing classes of semantic models for describing the behavior of Reo connectors. Although our presentation remains mainly informal, a similarly detailed overview does, to our knowledge, not exist.

In the second half of this paper, to take a step towards resolving Issue 2, we investigate the relation between two of the most important classes of semantic models of Reo connectors: (connector) coloring models (with two colors) [29] and constraint automata [17]. We show how to transform the former to the latter and demonstrate bisimilarity between an original and its transformation. In the opposite direction, we show how to transform constraint automata to equivalent coloring models, prove that these transformations define each other’s inverse, and again show bisimilarity. Additionally, we prove the compositionality of our transformation operators. To ensure that our transformation operators map one-to-one (instead of many-to-one), we extend coloring models with data-awareness.

A preliminary version of this paper, which excludes our comprehensive overview of semantic formalisms, appeared as [41].

Organization In Section 2, we give an informal description of Reo connectors; in Section 3, we provide an overview of semantic formalisms for describing their behavior. In Section 4, we extend coloring models with data-awareness. In Section 5, we discuss our transformation from coloring models to constraint automata; in Section 6, we discuss our transformation in the opposite direction. In Section 7, we discuss a potential application of our transformation operators. Section 8 concludes this paper.

2 Reo Connectors

A Reo connector consists of nodes (from the universe of nodes) through which data items (from the universe of data items) can flow.

Definition 1 (Universe of nodes). Node is the set of nodes.
Figure 1: Pictorial representation of Sync, LossySync and FIFO.

**Definition 2** (Universe of data items). Data is the finite set of data items.

Within a connector, we distinguish three types of nodes: input nodes on which components can perform write operations for data items, internal nodes that a connector uses to internally route data items, and output nodes on which components can perform take operations for data items. We call input and output nodes collectively, the boundary nodes of a connector. Write and take operations, collectively called i/o-operations, remain pending on a boundary node until they succeed, in which case the respective nodes fire. We call connectors without internal nodes, which form the most elementary mediums between components, primitives.

Figure 1 shows pictorial representations of three common (binary) primitives. The Sync primitive consists of an input node and an output node. Data items flow through this primitive only if both its nodes have pending i/o-operations. The LossySync primitive behaves similarly, but loses a data item if its input node has a pending write operation, while its output node has no pending take operation. We call LossySync a context-sensitive connector: depending on the presence or absence of pending i/o-operations on its nodes, i.e., its context, LossySync behaves differently—LossySync must never lose a data item if its output node has a pending take operation.\(^5\) Unfortunately, not all formalisms for modeling Reo connectors can describe context-sensitivity (at least, not directly); we address this topic in more detail in Section 3.

In contrast to the previous two primitives, connectors can have buffers to store data items in. Such connectors exhibit different states, while the internal configuration of Sync and LossySync always stays the same. For instance, the FIFO primitive consists of an input node, an output node, and a buffer of size 1. In its Empty state, a write operation on the input node of FIFO causes a data item to flow into its buffer—this buffer becomes full—while a take operation on its output node remains pending. Conversely, in its Full state, a write operation on its input node remains pending, while a take operation on its output node causes a data item to flow from the

\(^5\)Bonsangue et al. formalize context-sensitivity in [20, 21].
buffer to this output node—the buffer becomes empty.

We interpret \( n_1, n_2, \) and \( n_3 \) in Figure 1 as variables over nodes in Node: rather than depicting single connectors, this figure shows classes of connectors. For example, we can instantiate \textit{Sync} by setting \( n_1 \) to a concrete node \( A \) and \( n_2 \) to a concrete node \( B \). In text, we distinguish instances from classes by explicitly specifying the set of nodes an instance consists of. For example, \textit{Sync}(\( A, B \)) denotes a connector instance, while \textit{Sync} denotes a connector class. However, if no confusion can arise, we write “connector \textit{Conn}” rather than “instance of connector \textit{Conn}” for brevity.

We can construct complex connectors from simpler constituents by joining them. Informally, two connectors \textit{Conn} \(_1\) and \textit{Conn} \(_2\) can join iff, for each of their common nodes, this node serves as an input node in \textit{Conn} \(_1\) and as an output node in \textit{Conn} \(_2\) or vice versa.\(^6\)

To illustrate joining, Figure 2 shows the pictorial representation of \textit{LossyFIFO}, a connector composed of \textit{LossySync} and \textit{FIFO}. \textit{LossyFIFO} consists of one input node, one internal node, and one output node. Similar to \textit{FIFO}, \textit{LossyFIFO} exhibits the states Empty and Full. Informally, in the Empty state, a write operation on the input node of \textit{LossyFIFO} always causes a data item to flow into its buffer, while a take operation on its output node remains pending. In the Full state, a write operation on its input node always causes a data item to flow from its input node towards its buffer, but gets lost before reaching its internal node; a take operation on its output node causes a data item to flow from its buffer to this output node.

We call connectors that arise from joining smaller connectors \textit{compos-}

\(^6\)In this paper, we adopt the notion of nodes as introduced in [29]: at most two connectors can join on the same node. Alternatively [5], one can consider nodes entities on which an arbitrary number of connectors may coincide (rather than at most two). Under that interpretation, a node acts as a \textit{pumping station}: at each execution step, it accepts a data item on one of its input connectors and replicates this data item to all its output connectors. Moreover, in that view, connectors can join on non-boundary nodes. We favor [29]-style nodes, however, because of their simplicity. This choice does not affect the generality of our results, because we can implement [5]-style nodes with [29]-style nodes by joining ternary primitives (Merger and Replicator) that emulate this pumping station behavior [29].
ites. By hiding—another operation on connectors—the internal nodes of a composite \texttt{Conn}, abstracting these nodes from the definition of \texttt{Conn} such that the outside world cannot observe and interact with them anymore, we transform \texttt{Conn} to a primitive.

3 Overview of Semantic Formalisms

One can trace the history of Reo back to [3]: in that paper, Arbab informally introduces the basic concepts of Reo, while leaving a formalization of its semantics for future work. Indeed, as shown in Table 1, many formalization arose! In this section, we give an overview of these classes of models. We aim at comprehensiveness: to the best of our knowledge, we discuss all classes of semantic models that researches have developed in the context of Reo.

While we skip most of the formal definitions, we consider two semantic formalisms in more detail: \textit{constraint automata} in Section 3.2.1 and \textit{coloring models} in Section 3.3. This has two reasons. First, these two classes influenced and formed the basis of many other classes of models as well as implemented tools. Thus, these formalisms have played a crucial role in the research on the semantics of Reo. Second, in the subsequent sections of this paper, we prove the correspondence between these two classes, for which we need their formal definitions.

We continue this section as follows (see also Table 1). First, in Section 3.1, we discuss two classes of coalgebraic models. Second, in Section 3.2, we summarize operational models of Reo connectors. Third, in Section 3.3, we treat coloring models. Finally, in Section 3.4, we discuss those models that do not fit the previous three categories.

3.1 Coalgebraic Models

In the literature, we find two classes of coalgebraic models for describing the behavior of Reo connectors: those based on \textit{timed data streams} (TDS) and those based on \textit{record streams} (RS). In both these classes, the coalgebraic notion of \textit{streams} plays a prominent role. Informally, a stream $s$ over a set $S$ denotes an infinite sequence of elements from $S$. We denote the set of all the streams over $S$ by $S^\omega$. More formally, we define streams over $S$ as total functions from the natural numbers to elements in $S$, i.e., $s \in S^\omega$ iff $s : \mathbb{N} \to S$. We refer to the $i$-th element in a stream $s$ with the notation $s(i)$. 
Timed data streams   In [4, 5, 14, 62], Arbab and Rutten introduce TDS models as the first formalization of the semantics of Reo connectors. Informally, a TDS model of a connector describes for each of its nodes which and when—in dense time—data items flow through this node. It does so by associating each node with a *timed data stream* (TDS). We define a TDS as a pair of two streams over two different sets: a *data stream* \(d\) over `\(\text{DATA}\)` (the data domain; see Definition 2) and a monotonically increasing *time stream* \(t\) over \(\mathbb{R}_{\geq 0}\) (the positive real numbers including zero). Informally, a TDS of a node \(n\) conveys that data item \(d(i)\) passes through \(n\) at time \(t(i)\). By associating each node of a connector \(\text{Conn}\) with its own TDS in a TDS *tuple*, thus, we describe a single execution of \(\text{Conn}\). To describe all possible executions of \(\text{Conn}\), we associate \(\text{Conn}\) with a set of TDS tuples; we call such a set the TDS model of \(\text{Conn}\). Commonly, however, we define such a TDS model as a predicate on TDSs that induces the set of admissible TDS tuples of \(\text{Conn}\). (Enumerating all admissible TDS tuples of \(\text{Conn}\) becomes impossible because, not only each stream in a TDS itself is infinite, the set of admissible TDS tuples usually contains infinitely many elements.)

Record streams   The second class of coalgebraic models of Reo connectors contains models that, to describe a single execution of a connector \(\text{Conn}\), associate \(\text{Conn}\) with a single stream of *records* (cf., TDS models associate each node of \(\text{Conn}\) with a pair of streams). Izadi et al. introduce RS models in [38, 40]. Informally, records describe single execution steps of a connector: a record associates the nodes through which a data item flows (in the execution step it models) with those data items.

**Definition 3** (Record [40]). A record \(r\) is a partial function \(r : \text{NODE} \rightarrow \text{DATA}\) that maps a node \(n\) to a data item \(r(n)\). We denote the set of all records by \(\text{RECORD}\).

A record steam (RS) \(rs\) denotes a stream over the set `\(\text{RECORD}\)`.

A record steam (RS) \(rs\) denotes a stream over the set `\(\text{RECORD}\)`.

If the domain of every record in an RS \(rs\) includes only nodes from a connector \(\text{Conn}\), \(rs\) potentially describes a single execution of \(\text{Conn}\) (cf., a TDS tuple): if so, record \(rs(i)\) describes which data items flow through which nodes at an abstract time instant \(i\). To describe all executions of a connector \(\text{Conn}\), we associate \(\text{Conn}\) with a set of RSs, which we call its RS model.

Compared to TDS models, RS models differ in their disregarding of the exact arrival times of data items at nodes: RS models capture only the order in which data items arrive. In [38, 40], Izadi et al. also define an operator
for transforming TDS tuples to RSSs and an operator for transforming RSSs to TDS tuples. Furthermore, Izadi et al. assert that the latter operator forms the inverse of the former operator (but not vice versa).

3.2 Operational Models

Although stream-based models provide an intuitive way of thinking about flow through nodes, they turned out difficult to derive implementations from and analyze (e.g., by means of model checking). To remedy this, researchers looked for other types of models for describing the behavior of Reo connectors, including operational models. In fact, many such classes of operational models came to existence, e.g., (numerous variants of) constraint automata, various automata for describing context-sensitive connectors, and a structural operational semantics.

3.2.1 Constraint automata

In [10, 17], Baier et al. introduce the first class of operational models for describing the behavior of Reo connectors: constraint automata (CA). Similar to how ordinary automata accept strings, CA accept TDS-tuples.

Informally, a CA consists of a (possibly singleton) set of states, which correspond one-to-one to the states of the connector whose behavior it models, and a set of transitions between them; in contrast to ordinary automata, however, CA do not have accepting states. A transition of a CA, which describes a single execution step, carries a label that consists of two elements: a set of nodes and a data constraint. The former, called a firing set, describes which nodes synchronously fire in the state the transition leaves from; the latter specifies the conditions that the data items that flow through these firing nodes must satisfy.

**Definition 4** (Universe of data constraints [17]). \( DC(N) \) with \( N \subseteq \text{NODE} \) is the set of data constraints such that each \( dc \in DC(N) \) complies with the following grammar:

\[
dc ::= dc \land dc \mid dc \land \neg dc \mid T \mid \#n = d \text{ with } n \in N \text{ and } d \in \text{DATA}.
\]

Informally, \( \#n \) means “the data item that flows through \( n \),” while \( \land, \neg, \text{ and } T \) have their usual meaning. We also allow their derived Boolean operators.

---

7Because CA do not have accepting states, CA have only accepting runs of infinite length (similar to \( \omega \)-automata [63]). This property makes CA suitable for accepting TDS-tuples (because every TDS in such a tuple has an infinite length).
as syntactic sugar, e.g., ∨, and adopt \#n_1 = \#n_2 (with n_1, n_2 ∈ NODE) as an abbreviation of \( \bigvee_{d ∈ \text{DATA}} (\#n_1 = d ∧ \#n_2 = d) \). Below, we define CA.

**Definition 5** (Constraint automaton [17]). A constraint automaton CA over \([N ⊆ \text{NODE}, DC ⊆ \mathbb{D}(N)]\) is a triple \( \langle Q, T, q_0 \rangle \) with \( Q \) a set of states, \( T ⊆ Q × φ(N) × DC × Q \) a transition relation such that \( ⟨q, F, dc, q′⟩ ∈ T \) implies \( dc ∈ \mathbb{D}(F) \), and \( q_0 ∈ Q \) an initial state.

The condition “\( ⟨q, F, dc, q′⟩ ∈ T \) implies \( dc ∈ \mathbb{D}(F) \)” asserts that a data constraint cannot constrain data through nodes that do not fire. Henceforth, we consider only CA whose transition relations satisfy the following condition:

\[
⟨q, F_1, dc_1, q_1′⟩, ⟨q, F_2, dc_2, q_2′⟩ ∈ T \text{ and } q_1′ ≠ q_2′ \text{ implies } ⟨F_1, dc_1⟩ ≠ ⟨F_2, dc_2⟩
\]

This condition ensures that we can cast transition relations into transition functions,\(^8\) and it comes without loss of generality.\(^9\)

To illustrate Definition 5, Figure 3 shows the CA of Sync, LossySync, and FIFO. For simplicity, we assume the universe of data items a singleton with the string “foo” as its sole element. In general, the CA of FIFO contains a distinct state for each data item in DATA that may occupy its buffer. Note that rather than naming states symbolically (e.g., \( q, p, q_0, q_1, \ldots \)), we name states by meaningful indexes (below the states in font).\(^{10}\)

---

\(^8\)One may call CA satisfying this condition “syntactically deterministic.” This “syntactic determinism,” however, differs from determinism as originally defined for CA [17]; the latter takes into account logical equivalence between data constraints in transitions leaving the same state (instead of syntactic equivalence). This yields a stronger form of determinism. Thus, every “logically deterministic” CA is a “syntactically deterministic” CA. Conversely, every “syntactically nondeterministic” CA is a “logically nondeterministic” CA.

\(^9\)For every “logically nondeterministic” CA, there exists a language-equivalent “logically deterministic” CA [17]. Consequently, for every “syntactically nondeterministic” CA, there exists a “syntactically deterministic” CA that models the behavior of the same connector (at the level of granularity of language equivalence).

\(^{10}\)More precisely, Figure 3 shows classes of CA similar to how Figure 1 shows the pictorial representations of classes of connectors. An instance of a CA from Figure 3 contains actual nodes instead of the variables \( n_1 \) and \( n_2 \). Moreover, the indexes in such an instance convey which nodes this instance contains, e.g., \( \text{Sync}(A, B) \).
Definition 6 (Universe of indexes). \textit{INDEX} is the set of indexes.

Henceforth, without loss of generality, we assume \( Q \subseteq \text{INDEX} \) for all \( \text{CA} = \langle Q, T, q_0 \rangle \). Finally, unlike the original publications on \( \text{CA} \) (but similar to many later papers), we model the possibility of connectors to idle explicitly with the inclusion of self-transitions labeled by \( \langle \emptyset, \top \rangle \).\(^{11}\) This simplifies the definition of the join operator for \( \text{CA} \), below.

When we join two connectors \( \text{Conn}_1 \) and \( \text{Conn}_2 \) with \( \text{CA} \) as their behavioral model, we can compute the \( \text{CA} \) of the resulting composite by joining the \( \text{CA} \) of \( \text{Conn}_1 \) and \( \text{Conn}_2 \): the join operator for \( \text{CA} \) takes the Cartesian product of the sets of states of its arguments, designates the pair of their initial states as the initial state of the new \( \text{CA} \), and determines a new transition relation.

\[ \text{CA}_1 \Join \text{CA}_2 = \langle Q_1 \times Q_2, T, (q_0^1, q_0^2) \rangle \]

with:
\[
T = \left\{ \left( \begin{array}{c}
\langle q_1, q_2 \rangle, \\
F_1 \cup F_2, dc_1 \land dc_2, \\
\langle q_1', q_2' \rangle
\end{array} \right) \mid \begin{array}{c}
\langle q_1, F_1, dc_1, q_1' \rangle \in T_1 \\
\langle q_2, F_2, dc_2, q_2' \rangle \in T_2 \\
\text{and } F_1 \cap N_2 = F_2 \cap N_1
\end{array} \right\}.\]

The condition \( F_1 \cap N_2 = F_2 \cap N_1 \) in the previous definition asserts the following: if transitions in \( \text{CA}_1 \) and \( \text{CA}_2 \) form a new transition in \( \text{CA}_1 \Join \text{CA}_2 \), those transitions agree on the firing of the common nodes of \( \text{CA}_1 \) and \( \text{CA}_2 \). Furthermore, we remark that because \( \text{CA} \) abstract from the direction of flow, Definition 7 does not take into account the prerequisite in Section 2 that “connectors \( \text{Conn}_1 \) and \( \text{Conn}_2 \) can join iff, for each of their common nodes, this node serves as an input node in \( \text{Conn}_1 \) and as an output node in \( \text{Conn}_2 \) or vice versa.”

To illustrate Definition 7, Figure 4 shows the \( \text{CA} \) of \text{LossyFIFO}, obtained by joining the \( \text{CA} \) of \text{LossySync} and \text{FIFO} (see Figure 3). Note that this \( \text{CA} \) does not model the intended semantics of \text{LossyFIFO}: its transition \( \langle \text{LFIFO-E}, \{ n \}, \top, \text{LFIFO-E} \rangle \) describes the inadmissible loss of data in the case of an empty buffer. This example shows that \( \text{CA} \) cannot model context-sensitive connectors directly; in Section 3.2.3, we discuss operational formalisms that can.

\(^{11}\)Generally, transitions labeled by \( \langle \emptyset, \top \rangle \) serve as \( \tau \)-transitions, and they can occur also between two different states.

\(^{12}\)Henceforth, we write \( DC_1 \land DC_2 \) for \( \{ dc_1 \land dc_2 \mid dc_1 \in DC_1 \text{ and } dc_2 \in DC_2 \} \).
For more examples of CA, including nondeterministic CA, we refer to [10, 17]. Additionally, in [10, 17], Baier et al. define (bi)simulation for CA, they present the hide operators for CA, and they prove the compositionality of these operators under (bi)simulation. Furthermore, in [25, 26], Clarke introduces the forget operator for CA to model reconfiguration of connectors.

### 3.2.2 Variants of CA

Several variations on ordinary CA came to existence in the past five years. We start with three less intricate variants.

**Port automata** In [45], Koehler and Clarke introduce port automata (PA) by abstracting data constraints from CA. This means that a transition of a PA carries only a firing set (or, equivalently, a firing set and $\top$).

**CA with state memory** In [60], Pourvatan et al. introduce CA with state memory (CASM) by extending CA with a construct that enables data constraints to refer to the values of memory cells (e.g., the buffers of instances of FIFO). More formally, Pourvatan et al. associate a CASM not only with the usual ingredients of CA, but also with a set of memory cells and a value function. This latter function maps, for each state, memory cells to data items. Furthermore, Pourvatan et al. extend the syntax and semantics of data constraints with constructs for the formulation of propositions over memory cells.

**Labeled CA** In [44], Klüppelholz and Baier introduce labeled CA. In addition to the usual ingredients of CA, Klüppelholz and Baier associate each labeled CA with a set of propositions and a labeling function that maps each state to those propositions that hold in it.

---

13In fact, if $|\text{DATA}| = 1$, there exists a language equivalent PA for every CA.
We proceed with six more complex extensions: timed CA, probabilistic CA, continuous-time CA, quantitative CA, resource-sensitive timed CA, and transactional CA.

**Timed CA** In [8, 9], Arbab et al. introduce timed CA (TCA) for describing the behavior of time-dependent connectors, e.g., a variant of FIFO that loses a data item in its buffer after three time units.

In addition to the usual ingredients of CA, Arbab et al. include a set of clocks in each TCA: special variables used to register and evaluate constraints about the passage of time (cf., clocks in timed automata [2]). Similar to ordinary CA, the states of a TCA correspond one-to-one to the internal states of the connector it models. However, Arbab et al. associate each of those states also with a clock constraint, which asserts a condition on the values of the available clocks. Only as long as the clock constraint of a state holds, the TCA can remain in that state; otherwise, it must make a transition. The label on a transition in a TCA consists not only of a firing set and a data constraint, but also of a clock constraint cc and a reset set Reset. Informally, a transition \( \langle q, F, dc, cc, Reset, q' \rangle \) in a TCA describes an execution step of a connector in a state \( q \) before which the clock constraint \( cc \) holds and wherein data items that satisfy \( dc \) flow through the nodes in \( F \), bringing the connector in the successor state \( q' \) after resetting the clocks in Reset.

In [8, 9], Arbab et al. also define the join and hide operators for TCA and assert their compositionality under language equivalence.

**Probabilistic CA** In [15], Baier introduces probabilistic CA (PCA) for describing the behavior of probabilistic connectors, e.g., a probabilistic variant of LossySync or a randomized Sync.\(^{14}\)

Similar to ordinary CA, the states of a PCA correspond one-to-one to the internal states of the connector it models. The transition relation of a PCA, however, contains pairs that consist of: a state and a (discrete) probability distribution \( \Pi : \wp(N) \times \text{RECORD} \times Q \to [0, 1] \) over firing sets, records,\(^{15}\) and states. Informally, a transition \( \langle q, \Pi \rangle \) in a PCA describes an execution step of a connector in a state \( q \) wherein, with a probability \( \Pi(F, r, q') \), data items flow through the nodes in \( F \) according to \( r \), bringing the connector in the

\(^{14}\)A randomized Sync transforms data items flowing through its input node to other data items according to some probability distribution. See Example 4 in [15].

\(^{15}\)In [15] (and in other papers on CA and its variants), Baier calls records data assignments. Here, we use the terminology and notation of records for uniformity.
successor state \( q' \). In the same paper, Baier introduces an abstraction of PCA, called *simple probabilistic CA* (SPCA), whose transition relation contains quadruples that consist of: a state, a firing set, a data constraint, and a probability distribution \( \pi : Q \rightarrow [0, 1] \) over states. Informally, a transition \( \langle q, F, dc, \pi \rangle \) in an SPCA describes an execution step of a connector in a state \( q \) wherein data items that satisfy \( dc \) flow through the nodes in \( F \), bringing the connector in the successor state \( q' \) with probability \( \pi(q') \). Thus, PCA allow for a finer definition of the probability distribution than SPCA.

In [15], Baier also shows that one can transform any CA to an equivalent SPCA, and every SPCA to an equivalent PCA (and therefore, every CA to an equivalent PCA). Moreover, Baier defines the join and hide operators for SPCA and PCA, she defines bisimulation for PCA, and she asserts the compositionality of the join and hide operators for PCA under bisimulation.

**Continuous-time CA** In [18], Baier and Wolf introduce *continuous-time CA* (CCA) for describing both the behavior of connectors and their *time-dependent stochastic assumptions*, e.g., the stochastic waiting time of pending I/O-operations. This extension of CA enables a formal analysis of the performance of connectors. Although both TCA and CCA incorporate a notion of time, TCA model *functional* temporal aspects of connectors, while CCA model (some of) their *non-functional* temporal stochastic properties.

Similar to ordinary CA, the states of a CCA correspond one-to-one to the internal states of the connector it models. The transition relation of a CCA, however, has two partitions: one with ordinary CA transitions, called *interactive transitions* in the context of CCA, and a partition with *Markovian transitions*. This latter partition contains triples that consist of: a state \( q \), a *rate* \( \lambda \), and a successor state \( q' \). The rate, a positive real number, denotes the rate parameter of an exponential distribution over time (as in continuous-time Markov chains). Informally, a Markovian transition \( \langle q, \lambda, q' \rangle \) in a CCA describes the occurrence of a delay—whatever the source—of \( t \) or less time units in state \( q \) with probability \( 1 - e^{-\lambda t} \). A Markovian transition can fire only in the absence of enabled interactive transitions.

In [18], Baier and Wolf also define the join and hide operators for CCA. Moreover, Baier and Wolf define three variants of bisimulation for CCA—strong, weak, and very weak—and they assert the compositionality of the join and hide operators for CCA under strong and weak bisimulation.
Quantitative CA In [12, 53], Arbab et al. introduce quantitative CA (QCA) for describing both the behavior of connectors and the quality of service (QoS) guarantees they provide, e.g., reliability or shortest transmission time. From a conceptual perspective, QCA differ from TCA and PCA, because these latter two classes of models describe functional aspects of connectors, while QoS guarantees belong to the class of non-functional properties. The difference between QCA and CCA seems more subtle. On the one hand, QCA generalize CCA, because QCA can describe not only information about delays, but also about, e.g., reliability. On the other hand, CCA describe stochastic delays, which QCA seem unable to do.\footnote{To show that QCA can describe stochastic delays, one must show how to encode continuous probability distributions as Q-algebras [24].} Thus, the sets of connectors whose QoS guarantees QCA and CCA can describe overlap, but seem not coincident.

Similar to CA, the states of a QCA correspond one-to-one to the internal states of the connector it models. The label on a transition in a QCA, however, consists not only of a firing set and a data constraint, but also of a cost that represents a QoS metric (e.g., reliability). Informally, a transition \(\langle q, F, dc, c, q' \rangle\) in a QCA describes an execution step of a connector in state \(q\) wherein data items that satisfy \(dc\) flow through the nodes in \(F\), bringing the connector in the successor state \(q'\), while providing the QoS guarantees described by \(c\). Formally, a cost \(c\) comes from the domain of a \(Q\)-algebra—an algebraic structure, introduced by Chothia and Kleijn in [24], for the compositional description of QoS properties.

In [12], Arbab et al. also define the join and hide operators for QCA. Moreover, Arbab et al. define four types of simulation for QCA—strong, weak, and (weak) quality improving—and they prove the compositionality of the join and hide operators for QCA under strong, weak, and quality improving simulation. One can use quality improving simulations to analyze, e.g., if an implementation of a connector provides a higher QoS than its specification.

Resource-sensitive timed CA Concurrently with QCA, in [51], Sun Meng and Arbab introduce resource-sensitive timed CA (RSTCA) for describing the behavior of time-dependent resource-sensitive connectors, e.g., a connector that requires a sufficiently large bandwidth and that times out if this resource remains unavailable during some interval. The class of RSTCA differs from the class of CCA, because CCA incorporate a stochastic notion of the passage of time, while RSTCA incorporate crisp values. Compared to TCA, RSTCA seem more restrictive in the sense that an RSTCA does not feature explicit
clocks: instead, each state $q$ and each transition $t$ in an RSTCA has its own implicit clock, which this RSTCA resets to 0 once it reaches $q$ or fires $t$.

In addition to the usual ingredients of CA, Sun Meng and Arbab include a set of resources in each RSTCA: special variables used to register information and evaluate constraints about resources (e.g., available bandwidth). Similar to ordinary CA, the states of an RSTCA correspond one-to-one to the internal states of the connector it models. The transition relation of an RSTCA, however, has two partitions: one with interactive transitions (different from the interactive transitions in CCA) and another with timeout transitions.

The former partition contains transitions with a label that consists not only of a firing set and a data constraint, but also of a resource constraint $rc$ and a duration function $df$. Resource constraints assert a condition about the required resources, while duration functions map resources to those times at which a connector must have finished using them. Informally, an interactive transition $\langle q, F, dc, rc, df, q' \rangle$ in an RSTCA describes an execution step of a connector in state $q$ wherein data items that satisfy $dc$ flow through the nodes in $F$, while utilizing a set of resources $R$ that satisfy $rc$, bringing the connector in the successor state $q'$ within time $df(R)$.

The partition with timeout transitions contains triples that consist of: a state $q$, a timeout $t$, and a successor state $q'$. Informally, a timeout transition $\langle q, t, q' \rangle$ describes an execution step of a connector in state $q$ wherein it transits to the successor state $q'$, if no interactive transitions could fire for $t$ time units. Timeout transitions in QCA resemble Markovian transitions in CCA, but carry a crisp maximum value rather than a stochastic parameter.

In [51], Sun Meng and Arbab also define the join and hide operators for RSTCA. Moreover, Sun Meng and Arbab define two types of simulation for RSTCA—functional and strong—and prove the compositionality of the join and hide operators for RSTCA under strong simulation.

**Transactional CA** In [54], Sun Meng and Arbab introduce transactional CA (TNCA) for describing the behavior of connectors whose execution involves *long-running transactions*. Such transactions often provide weaker guarantees than the traditional ACID guarantees for transactions [36] (atomicity, consistency, isolation, durability): for instance, because data involved in long-running transactions is not locked, such transactions typically lack isolation. Long-running transactions occur in service-oriented architectures, thus when using Reo in this application domain, support for long-running transactions becomes important.
Similar to CA, the states of a TNCA correspond one-to-one to the internal states of the connector it models. The label on a transition in a TNCA, however, consists of a set of nodes and either a data constraint or a transaction $\psi$. In the former case, we obtain a transition as in ordinary CA, called an atomic transition in the context of TNCA, and we interpret the set of nodes this transition carries as a firing set. In the latter case, in contrast, we have a transaction transition. Informally, a transaction transition $\langle q, F, \psi, q' \rangle$ in a TNCA describes an execution step of a connector in state $q$ wherein the nodes in $F$ (commit to) participate in the transaction $\psi$, bringing the connector in the successor state $q'$. As transactions themselves typically involve interactions and coordination among multiple parties, Sun Meng and Arbab model transactions as Reo connectors.

In [54], Sun Meng and Arbab also define the join operator for TNCA.

### 3.2.3 Context-Sensitive Automata

As mentioned during our discussion of LossyFIFO in Section 3.2.1, there exists a significant class of connectors whose behavior CA cannot describe satisfactorily: context-sensitive connectors.\footnote{Later in this paper, however, we demonstrate that CA actually can describe the behavior of context-sensitive connectors.} We recall that the behavior of a context-sensitive connector depends not only on its own internal state, but also on the presence or absence of pending i/o-operations on its boundary nodes, e.g., a LossySync should lose a data item only if it has no pending take operation on its output node. Unfortunately, CA can directly model only a nondeterministic approximation of LossySync (as in Figure 3). Below, we discuss other classes of automaton-based models that can describe the behavior of context-sensitive connectors.

**B"uchi automata** In [38, 40], Izadi et al. propose to use B"uchi automata of records (BAR), i.e., automata on infinite words \cite{sun} over records (see Definition 3), for describing the behavior of connectors. Similar to how CA accept TDSS, BAR accept RSS, i.e., record streams. Moreover, the states of a BAR (in its simple form) correspond one-to-one to the internal states of the connector Conn it models, while its transitions, each labeled with a record, describe the execution steps of Conn. Because BAR feature (sets of) accepting states, called (generalized) acceptance conditions, BAR can model connectors with fairness constraints (e.g., if a node can fire from some instant onwards, it eventually does so), whose behavior CA cannot describe.
For describing the behavior of context-sensitive connectors, in [39, 40], Izadi et al. introduce *augmented* BAR (ABAR), whose states do no longer correspond one-to-one to the internal states of connectors, but *many*-to-one: each state of an ABAR does not only represent the internal state of the connector Conn it models, but also registers information about the presence or absence of I/O-operations on the nodes in Conn. While BAR accept infinite sequences of records, i.e., RSS, ABAR accept *alternating* infinite sequences of (i) sets of nodes that have pending I/O-operations, and (ii) records, describing what data items actually flow through which of those nodes.

In [38, 40], Izadi et al. also define the join and hide operators for BAR (the join operator for BAR differs from the usual product operator for Büchi automata). Moreover, Izadi et al. define an operator for transforming an arbitrary CA to a BAR and prove its compositionality (in [40]). Similarly, in [39, 40], Izadi et al. define the join and hide operators for ABAR.

**Guarded automata** In [20, 21], Bonsangue et al. define *guarded automata* (GA) for describing the behavior of (context-sensitive) connectors.\(^\text{18}\) Similar to Izadi’s ABAR, GA accept alternating infinite sequences of *guards* and firing sets. We can consider these guards generalizations of the sets of nodes that appear in the infinite words that ABAR accept—guards assert a condition about the presence or absence of I/O-operations on nodes—while we can consider these firing sets abstractions of the records that appear in the words that ABAR accept (because firing sets exclude information about data).

Similar to CA, but in contrast to ABAR, the states of a GA correspond one-to-one to the internal states of the connector it models: information about the presence or absence of I/O-operations does not appear in the states of a GA (as in ABAR), but as guards on its transitions. Transitions additionally carry a firing set, i.e., the nodes through which a data item flows if the corresponding guard holds and the transition fires. Unlike ABAR, GA do not have accepting states.

In [20, 21], Bonsangue et al. also define the join operator for GA (although indirectly, i.e., in terms of two other operators, namely product and synchronization). Moreover, Bonsangue et al. define bisimulation for GA, and assert the compositionality of the join operator under bisimulation. Also, in [20], Bonsangue et al. cast port automata (PA—see Section 3.2.2) and CA into GA, which yields context-sensitive variants of PA and CA.

---

\(^\text{18}\)Bonsangue et al. use also the name *Reo automata* in [20, 21], but because Costa uses this name to refer to a different class of automata in [33], we write GA to avoid confusion.
In [56, 57], Moon et al. extend ordinary GA to stochastic GA (SGA) by (i) associating each node with an arrival rate that describes, stochastically, the rate at which I/O-operations arrive, and (ii) associating each transition \( t \) with information about the stochastic delay of flow through the nodes that participate in \( t \) (similar to the rate on a Markovian transition in a CCA). In [56, 57], Moon et al. also define the join operator for SGA (indirectly in terms of product and synchronization).

**Intentional automata** In [33], Costa introduces *intentional automata* (IA) for describing the behavior of (context-sensitive) connectors.\(^{19}\)

Similar to Izadi’s ABAR, the states of an IA correspond many-to-one to the internal states of the connector it models: each state of an IA not only captures such an internal state, but also registers the presence of I/O-operations. The classes of ABAR and IA differ, however, in the type of labels their transitions carry: in ABAR, transitions carry records, while in IA, transitions carry pairs that each consist of a firing set and another set of nodes. Such a second set of nodes on a transition \( t \) contains those nodes on which an I/O-operation becomes pending during the execution step \( t \) describes. Similar to GA, but unlike ABAR, IA do not have accepting states.

In [33], Costa also defines the join and hide operators for IA. Moreover, Costa defines (weak) bisimulation for IA, and proves the compositionality of the join and hide operators for IA under weak bisimulation.

In [13], Arbab et al. extend ordinary IA to *quantitative IA* (QIA) by extending the labels on transitions in ordinary IA with (i) a data constraint (as in CA) and (ii) information about the stochastic delay and arrival rates of I/O-operations and data items on the nodes that participate in a transition. In [13], Arbab et al. also define the join operator for QIA (although indirectly, i.e., in terms of two other operators, namely product and refinement).

**Action CA** In [46], Kokash et al. introduce *action CA* (ACA) for describing the behavior of (context-sensitive) connectors and their temporal QoS properties *without* incorporating an explicit notion of time. Instead, Kokash et al. model delays through the successive execution of distinct *actions* such as “block node \( n \)” and “unblock node \( n \).” Thus, despite their name, ACA differ significantly from ordinary CA and other CA-based models for describing the temporal QoS aspects of connectors (i.e., CCA, QCA, and RSTCA).

---

\(^{19}\)Costa uses also the name *Reo automata* in [33], but because Bonsangue et al. use this name to refer to a different class of automata in [20, 21], we write IA to avoid confusion.
Similar to Izadi’s ABar and Costa’s IA, the states of an ACA correspond many-to-one to the internal states of the connector it models and, among other information, can register the presence or absence of I/O-operations. The transition relation of an ACA contains quadruples that consist of: a state, a set of actions over a set of nodes $N$, a data constraint (as in ordinary CA), and a successor state. An action in the set of actions that a transition $t$ carries denotes an operation (e.g., blocking/unblocking nodes from accepting I/O-operations or starting/stopping the propagation of data between nodes) that a connector performs in the execution step $t$ describes. In the subclass of ACA wherein each action has the form “propagate a data item through node $n$,” sets of actions reduce to firing sets, and ACA simplify to CA.

In [46], Kokash et al. also define the join and hide operators for ACA. Moreover, Kokash et al. encode the ACA of seven primitives into the process algebra mCRL2. Shortly, in Section 3.4, we discuss mCRL2 models for Reo connectors in more detail.

Behavioral automata In [61], Proença introduces behavioral automata (BA) for modeling the behavior of connectors in a step-wise manner. Proença uses BA in [61] as the basis for an implementation of Reo and to justify the assumptions that underlie his Dreams framework, a distributed implementation of Reo. Because BA can embed other classes of semantic models—including GA—BA can describe the behavior of context-sensitive connectors.

Similar to CA and GA, the states of a BA correspond one-to-one to the internal states of the connector it models.

The transitions of a BA carry abstract labels, which map to quintuples $\langle N, F, I, O, data \rangle$ that describe atomic execution steps of a connector. In such a quintuple, $N$ denotes a set of nodes, $F \subseteq N$ denotes a firing set, $I \subseteq F$ and $O \subseteq F$ denote the input and output nodes involved, and $data$ denotes a function from $I \cup O$ to $\text{Data}$. Depending on the specific instantiation of a BA (Proença instantiates BA, among other classes of models, for CA and GA in [61]), some components in these quintuples remain unused. For instance, in the instantiation for GA, $I = O = data = \emptyset$.

Furthermore, Proença associates each state of a BA with a concurrency predicate. Essentially, a concurrency predicate in a BA $\text{BA}$ for a state $q$ describes a set of transitions in a BA different from $\text{BA}$ that can execute concurrently while $\text{BA}$ remains in $q$. More formally, a concurrency predicate $C$ for a state $q$ (in $\text{BA}$) denotes a set of labels such that $l \in C$ implies that transitions (outside $\text{BA}$) labeled by $l$ can execute concurrently.
In [61], Proença also defines the join operator for BA.

### 3.2.4 Structural Operational Semantics

Finally, in [58], Mousavi et al. formalize the semantics of (some of the primitives in) Reo with a structural operational semantics (SOS) in Plotkin’s style [59]. For each primitive, Mousavi et al. define a set of transition rules that induce a transition system. The states in this transition system consist of pairs \(\langle Sys, Val \rangle\) with \(Sys\) a Reo system term (RST) and \(Val\) a function that maps nodes to (possibly infinite) sequences of data items. An RST describes the structure of a Reo connector \(Conn\), including the specific primitives that \(Conn\) consists of, while the sequences of data items to which \(Val\) maps a node \(n\) represent those data items that in some instant already have arrived at \(n\). Supplemented with the maximal progress assertion of Khosravi et al. in [43], the SOS semantics can model context-sensitive connectors.

In [58], Mousavi et al. also define special transition rules for joining two connectors—these rules correspond to the join operators defined for other classes of semantic models of Reo connectors. Moreover, Mousavi et al. define (bi)simulation for the transition systems that their transition rules induce.

### 3.3 Coloring Models

We proceed with a third kind of semantic formalisms: (connector) coloring models (CM), introduced by Clarke et al. in [28, 29] and later extended by Costa in [33]. Coloring models work by marking the nodes of a connector with colors that specify whether data items flow through these nodes or not. Depending on the number of colors, different models with different levels of expressiveness arise. In Section 3.3.1, we discuss CMs with three colors. For now, we assume a total of two colors, which yields 2-coloring models (2CM): the flow color (data items can flow through the nodes it marks) and the no-flow color (data items cannot flow through the nodes it marks).

**Definition 8** (Colors [29]). \(\text{COLOR} = \{\text{-----}, \text{-----}\}\) is a set of colors.

To describe an execution step of a connector, CMs consist of colorings: maps from a set of nodes to a set of colors, which assign to each node in the

\[^{20}\text{Including Sync, LossySync, and FIFO.}\]
set a color that indicates whether this node fires (or not) in the execution step the coloring describes. We collect all such possible execution steps of a connector in sets of colorings, called coloring tables.

**Definition 9** (Coloring [29]). A coloring $c$ over $N \subseteq \text{Node}$ is a function $c : N \to \text{Color}$ that maps a node $n$ to a color $c(n)$. We denote the set of all colorings over $N$ by $\text{Col}(N)$.

**Definition 10** (Coloring table [29]). A coloring table $T$ over $N \subseteq \text{Node}$ is a set $T \subseteq \text{Col}(N)$ of colorings over $N$.

To accommodate connectors that exhibit different behavior in different states (e.g., connectors with buffers), CMS feature coloring table maps (CTM): maps from a set of indexes (representing the states of a connector as in CA; see Definition 6) to a set of coloring tables (describing the admissible execution steps in these states). To model the change of state a connector incurs when (some of) its nodes fire, CMS feature next functions. The next function of a connector maps an index $i$ in the domain $I$ of a CTM $S$ and a coloring in the coloring table to which $S$ maps $i$ to an index in $I$ (possibly the same $i$).

**Definition 11** (CTM [33]). A CTM $S$ over $[N \subseteq \text{Node}, I \subseteq \text{Index}]$ is a function $S : I \to \wp(\text{Col}(N))$ that maps an index $i$ to a coloring table $S(i)$ over $N$.

**Definition 12** (Next function [33]). Let $S$ be a CTM over $[N, I]$. A next function $\eta$ over $S$ is a partial function $I \times \text{Col}(N) \rightarrow I$ that maps every [index, coloring]-pair $(i, c)$ such that $c \in S(i)$ to an index $\eta(i, c)$.

Finally, we define the CM of a connector Conn as a pair that consists of a next function (describing the behavior of Conn) and an index (representing its initial state).

**Definition 13** (CM). Let $S$ be a CTM over $[N, I]$. A CM $CM$ over $S$ is a pair $CM = (\eta, i_0)$ with $\eta$ a next function over $S$ and $i_0 \in I$.

To illustrate the previous definitions, Figure 5 shows the 2CMSs of Sync, LossySync, and FIFO, whose structures we depicted in Figure 1.

When we join two connectors Conn$_1$ and Conn$_2$ with CMS as their behavioral model, we can compute the CM of the resulting composite by joining the CMS of Conn$_1$ and Conn$_2$. We describe this joining process in a

---

$^{21}$In [33], Costa calls CTMs indexed sets of coloring tables.
bottom-up fashion. First, to join two compatible colorings—colorings that assign the same colors to their common nodes—we merge the domains of these colorings and map each node $n$ in the resulting set to the color that one of the colorings assigns to $n$. The join of two coloring tables comprises the computation of a new coloring table that contains the pairwise joins of the compatible colorings in the two individual coloring tables.

**Definition 14** (Join of colorings [29]). Let $c_1$ and $c_2$ be colorings over $N_1$ and $N_2$ such that $c_1(n) = c_2(n)$ for all $n \in N_1 \cap N_2$. Their join, denoted by $c_1 \cup c_2$, is a coloring over $N_1 \cup N_2$ defined as:

$$c_1 \cup c_2 = \left\{ n \mapsto \kappa \mid n \in N_1 \cup N_2 \text{ and } \kappa = \begin{cases} c_1(n) & \text{if } n \in N_1 \\ c_2(n) & \text{otherwise} \end{cases} \right\}.$$ 

**Definition 15** (Join of coloring tables [29]). Let $T_1$ and $T_2$ be coloring tables over $N_1$ and $N_2$. Their join, denoted by $T_1 \cdot T_2$, is a coloring table over $N_1 \cup N_2$ defined as:

$$T_1 \cdot T_2 = \left\{ c_1 \cup c_2 \mid c_1 \in T_1 \text{ and } c_2 \in T_2 \text{ and } c_1(n) = c_2(n) \text{ for all } n \in N_1 \cap N_2 \right\}.$$ 

The join of two CTMs comprises the computation of a new CTM that maps each pair of indexes in the Cartesian product of the domains of the two individual CTMs to the join of the coloring tables to which these CTMs map the indexes in the pair. We define the join of two next functions in terms of the Cartesian product, the join of colorings, and the join of CTMs.
Definition 16 (Join of CTMs [33]). Let $S_1$ and $S_2$ be CTMs over $[N_1, I_1]$ and $[N_2, I_2]$. Their join, denoted by $S_1 \circ S_2$, is a CTM over $[N_1 \cup N_2, I_1 \times I_2]$ defined as:

$$S_1 \circ S_2 = \{(i_1, i_2) \mapsto S_1(i_1) \cdot S_2(i_2) \mid i_1 \in I_1 \text{ and } i_2 \in I_2\}.$$ **Figure 6**: Colorings, $ctm$, and next function of LossyFIFO. Let LFIFO-E denote $(LSync, FIFO-E)$, and let LFIFO-F denote $(LSync, FIFO-F)$.

$$S = \{LFIFO-E \mapsto \{c_{12}, c_{24}, c_{44}\}, LFIFO-F \mapsto \{c_{23}, c_{24}, c_{44}\}\}$$

$$\eta = \begin{cases} (LFIFO-E, c_{12}) \mapsto LFIFO-F, & (LFIFO-F, c_{23}) \mapsto LFIFO-E, \\
(LFIFO-E, c_{24}) \mapsto LFIFO-E, & (LFIFO-F, c_{44}) \mapsto LFIFO-F, \\
(LFIFO-E, c_{44}) \mapsto LFIFO-E, & (LFIFO-F, c_{44}) \mapsto LFIFO-F. 
\end{cases}$$

$$i_0 = LFIFO-E$$

Let $\eta_1$ and $\eta_2$ be next functions over $S_1$ and $S_2$ defined over $[N_1, I_1]$ and $[N_2, I_2]$. Their join, denoted by $\eta_1 \otimes \eta_2$, is a next function over $S_1 \circ S_2$ defined as:

$$\eta_1 \otimes \eta_2 = \left\{ (i_1, i_2), c_1 \cup c_2 \mapsto \langle \eta_1(i_1, c_1), \eta_2(i_2, c_2) \rangle \mid (i_1, i_2) \in I_1 \times I_2 \text{ and } c_1 \cup c_2 \in (S_1 \circ S_2)(\langle i_1, i_2 \rangle) \right\}.$$ Finally, we define the join of CMSs in terms of the join of next functions and take the pair of the initial states as the initial state of the join.

Definition 18 (Join of CMSs). Let $CM_1 = \langle \eta_1, i^1_0 \rangle$ and $CM_2 = \langle \eta_2, i^2_0 \rangle$ be CMSs over $S_1$ and $S_2$. Their join, denoted by $CM_1 \bowtie CM_2$, is a CM over $S_1 \circ S_2$ defined as:

$$CM_1 \bowtie CM_2 = \langle \eta_1 \otimes \eta_2, \langle i^1_0, i^2_0 \rangle \rangle.$$ To illustrate the previous definitions, Figure 6 shows the 2CM of LossyFIFO, whose structure we depicted in Figure 2. In this figure, the index of a coloring specifies its origin: a coloring $c_i$ results from joining $c_i$ (of LossySync) with $c_j$ (of FIFO) in Figure 5. Note that this 2CM does not model the intended behavior of LossyFIFO: its coloring $c_{24}$ describes the
inadmissible loss of data in the case of an empty buffer. Thus, as CA, 2CMS seem incapable of modeling context-sensitive connectors. Shortly, we revisit this topic.

The introduction of CMS for Reo stood at the basis of several new tools—now part of the ECT—for animation, simulation, and verification of Reo connectors.

### 3.3.1 Three colors

In the same publications [28, 29] as those in which Clarke et al. introduce 2CMS, they introduce CMS with three colors, i.e., 3-coloring models (3CM). The difference between 2CMS and 3CMS lies in the instantiation of COLOR: when using 3CMS, we replace the no-flow color - - - - with the two different no-flow colors - - - - - and - - - - - - - . Although both these colors indicate that the nodes they mark have no flow, they also provide information about the reason for this absence: informally, we can think of the former color as stating that the nodes it marks require a reason, while we can think of the latter color as stating that the nodes it marks provide a reason (e.g., no pending i/o-operation). Apart from this different instantiation of COLOR, however, nothing changes: Definitions 9–13 and 14–18 remain the same.

The availability of two no-flow colors allows 3CMS, in contrast to 2CMS, to properly model context-sensitive connectors. For instance, in the case of LossySync, its 3CM asserts that this connector can lose a data item only if its output node receives a reason for an absence of flow by means of the following coloring (which replaces c2 in Figure 5):

\[
\{n_1 \leftrightarrow - , n_2 \leftrightarrow - - - - - \}.
\]

The potential of CMS to satisfactorily model context-sensitivity formed the main reason for their investigation and introduction. At the time of the first publications on CMS, two colors seemed insufficient for this, while three appeared adequate. In [42], however, its authors present an information-preserving mapping from 3CMS to 2CMS, which demonstrates that 2CMS actually can describe the behavior of context-sensitive connectors. This works as follows.

Usually, in 2CMS and 3CMS, one represents every conceptual node—a node that one would draw in a Reo diagram—with a single concrete node.

\footnote{One may, however, use an optimization called the flip-rule to collapse large coloring tables into smaller ones without loss of information. In that case, Definition 14 becomes slightly more complex. We refer to [29, 33] for details.}
in colorings. With two colors, subsequently, one can attribute two different
behavior alternatives to each conceptual node, e.g., flow or no-flow. However,
if we represent every conceptual node with two concrete nodes in colorings,
we can attribute \(2^2\) different behavior alternatives to each conceptual node.
Thus, to encode a 3CM as a 2CM, one should double the number of concrete
nodes in the 3CM and consistently map every color in the 3CM to a pair
of colors in the 2CM. Interestingly, such an encoding of three colors using
two concrete nodes and two colors also has an intuitive meaning [42]. More
generally, one can attribute \(2^k\) behavior alternatives to a conceptual node
represented by \(k\) concrete nodes and using two colors.

Because we can transform any CM with \(k \geq 3\) colors to a CM with only
two colors, we consider only 2CMs in (most of) the rest of this paper.

3.3.2 Tile models

In [11] (based on [22]), Arbab et al. introduce tile models for describing the
behavior of Reo connectors by casting CMS of connectors into tiles. Tiles,
introduced in [34], extend Plotkin-style inference rules: they describe tran-
sitions from an initial configuration (if some trigger goes off) to a final
configuration (producing some effect). Different from such inference rules,
however, one can compose tiles in three different ways: horizontally (synchron-
ization), vertically (composition in time), and in parallel (concurrency).

In [11], Arbab et al. consider Reo connectors hypergraphs and formalize
these as morphisms in a symmetric monoidal category. Subsequently, these
morphisms serve as configurations on which tiles operate: Arbab et al.
describe the behavior of connectors by associating these morphisms with tiles,
each of which describes one of its possible execution steps. More precisely,
Arbab et al. first define a series of tiles for many common primitives based
on their semantics in terms of 2CMs, prove the correctness of these tiles
(with respect to the corresponding 2CMs), and assert the compositionality
of these tiles with respect to horizontal and parallel composition. Second,
Arbab et al. define a series of tiles for a subset of these common primitives
based on their semantics in terms of 3CMs and assert the correctness (with
respect to the corresponding 3CMs) and compositionality of these tiles. Note
that this second series of tiles, i.e., those derived from 3CMs, can model
context-sensitive connectors.

Although, as remarked in [61], tile models comprise one of the most
complete semantic models of Reo connectors, no tools support them.
3.4 Other Models

**Process algebra**  In [47, 48, 49], Kokash et al. define the semantics of Reo connectors in terms of *processes* of the process algebra mCRL2 [35], an algebra that extends the process algebra ACP [19] with data and time. Kokash et al. propose two translations: one based on CA (extended with a TCA-style notion of time in [48]) and another based on 3CMs. In [49], Kokash et al. prove the correctness and compositionality of their translation from CA to mCRL2 specifications and assert similar results for their translation from 3CMs to mCRL2. The ECT includes an implementation of this work.

Because analysis tools, including a model checker, exist for mCRL2 specifications, we can use the translation from Reo to mCRL2 for verification of Reo connectors. In fact, the translation of a 3CM of a connector to its corresponding mCRL2 specification formed the only way to verify context-sensitive connectors until recently.\(^{24}\)

**Constraints**  In [30, 31, 32], Clarke et al. define the semantics of Reo connectors in terms of propositional *constraints* (but different from the data constrains in CA). Initially, in [31], Clarke et al. define each connector in terms of four different kinds of constraints: *synchronization constraints*, *data flow constraints*, *state constraints*, and *external constraints*. Synchronization constraints describe which nodes can synchronize in some execution step, while data flow constraints describe what particular data items flow in such a step through which nodes. State constraints describe how the state of a connector evolves during its execution, while external constraints capture externally maintained state. In [32], Clarke et al. prove the correctness and compositionality of this constraint-based approach with respect to CA.

In [32], Clarke et al. extend their initial approach with *context constraints* for modeling context-sensitive connectors. Similar to how Kokash et al. take 3CMs as the basis for their encoding of context-sensitive connectors into mCRL2, Clarke et al. base their context constraints on colorings in 3CMs. Moreover, Clarke et al. prove the correctness of this context-sensitive constraint-based approach with respect to 3CMs.

In [61], Proença uses SAT-solvers to execute connectors based on their constraint model, and he shows that this approach significantly improves

\(^{23}\) [http://www.mcrl2.org]

\(^{24}\) Another model checker for Reo, called Vereofy [16], operates on CA, which many consider incapable of modeling context-sensitivity (see also Section 3.2.1). We discuss Vereofy and its ability to model check context-sensitive connectors in Section 7.
performance (as compared to the execution of a connector based on its CM).

**Petri nets and intuitionistic temporal linear logic** In [27], Clarke defines the semantics of nine Reo connectors (including common primitives such as Sync, LossySync, and FIFO) in terms of zero-safe nets [23], a special class of Petri nets. Different from ordinary Petri nets, zero-safe nets feature two types of places: zero places and stable places. When moving from one marking to the next, a zero-safe net fires until all tokens occupy only stable places. Such an epoch may consist of multiple firings, but those intermediate markings wherein some tokens occupy zero places remain unobservable.

When modeling a Reo connector Conn with a zero-safe net, every epoch of this net represents a single execution step. In Clarke’s encoding, zero places ensure that the nodes in Conn cannot fire multiple times during a single epoch; otherwise, this would correspond to nodes of Conn having data flow more than once in a single execution step of Conn, which cannot happen. Other comparisons between Reo and Petri nets appear in [7, 50].

In [27], Clarke also casts Reo and zero-safe nets into intuitionistic temporal linear logic. Clarke does this not only for reasoning about coordination models, but also to enable more refined behavioral descriptions of connectors (e.g., to distinguish internal from external choices).

**Unifying theories of programming** In [52], Sun Meng and Arbab define the semantics of Reo connectors in terms of the unifying theories of programming (UTP) [37]. The UTP can provide a formal semantics for various languages, and thus, facilitates the specification of similar features in different languages in a similar style. This enables a straightforward comparison and analysis of different languages in the same framework.

Sun Meng and Arbab model Reo connectors as UTP designs. A UTP design comprises a pair of predicates of the form $P \vdash Q$ with $P$ an assumption on the input (of a connector) and $Q$ a commitment about the output (of a connector). The predicates $P$ and $Q$ induce sets of admissible tuples of timed data sequences: structures that resemble tuples of timed data streams (see Section 3.1), but which can have a finite length.

In [55], Sun Meng et al. use the UTP semantics of Reo for model based testing through refinement and test case generation (also based on the work presented in [1]).
Figure 7: Known relations between semantic formalisms. An arrow from formalism $X$ to formalism $Y$ means: if one can model the behavior of a connector $Conn$ in $X$, one can model the behavior of $Conn$ in $Y$ without loss of information.

$2\text{CM}$ : Coloring models with two colors [28, 29, 33]  
$3\text{CM}$ : Coloring models with three colors [28, 29, 33]  
$\text{ABAR}$ : Augmented BAR [39, 40]  
$\text{ACA}$ : Action CA [46]  
$\text{BA}$ : Behavioral automata [61]  
$\text{BAR}$ : Büchi automata of records [38, 40]  
$\text{CA}$ : Constraint automata [10, 17]  
$\text{CCA}$ : Continuous-time CA [18]  
$\text{Constr.}$ : Propositional constraints [30, 31, 32]  
$\text{GA}$ : Guarded automata [20, 21]  
$\text{IA}$ : Intentional automata [33]  
$\text{ITLL}$ : Intuitionistic temporal linear logic [27]  
$\text{LCA}$ : Labeled CA [44]  
$\text{mCRL2}$ : Process algebra [47, 48, 49]  
$\text{PA}$ : Port automata [45]  
$\text{PCA}$ : Probabilistic CA [15]  
$\text{QCA}$ : Quantitative CA [12, 53]  
$\text{QIA}$ : Quantitative IA [13]  
$\text{RS}$ : Record streams [38, 40]  
$\text{RSTCA}$ : Resource-sensitive timed CA [51]  
$\text{SGA}$ : Stochastic GA [56, 57]  
$\text{SOS}$ : Structural operational semantics [58]  
$\text{SPCA}$ : Simple PCA [15]  
$\text{TCA}$ : Timed CA [8, 9]  
$\text{TDS}$ : Timed data streams [4, 5, 14, 62]  
$\text{Tiles}$ : Tile models [11]  
$\text{TNCA}$ : Transactional CA [54]  
$\text{UTP}$ : Unifying theories of programming [55, 52]  
$\text{ZSN}$ : Zero-safe nets [27]
3.5 Summary

Figure 7 summarizes the relations between the semantic formalisms discussed in this section.

4 Data-Aware Coloring Models

As demonstrated in the previous section, constraint automata and coloring models influenced and formed the basis of many other classes of semantic models: CA inspired at least TCA, PCA, CCA, QCA, RSTCA, TNCA, while CMS inspired tile models, the mCRL2 models, and the models based on constraints. Formally establishing a correspondence between CA and CMS, therefore, has great value: it paves the way for the application of tools and extensions devised for CA to CMS and vice versa.

In this section, we take a first step towards this goal: we make traditional CMS data-aware by extending them with constraints similar to those carried by transitions in CA. We introduce this extension, because one of the transformation operators that we define later in this paper lacks a desirable property otherwise: it would map many-to-one instead of one-to-one. More precisely, the transformation from CA to CMS would map different CA, namely those whose transitions carry different data constraints but equal firing sets, to the same CM. Alternatively, to gain this one-to-one property, we could have narrowed our scope to PA, which abstract from data constraints (see Section 3.2.2). We favor an extension of CMS for generality.

We add data-awareness to CMS, independently of the number of colors, by associating each coloring with a data constraint. Such a constraint coloring describes an execution step of a connector wherein data items that satisfy the data constraint flow through the nodes marked by the flow color. Below we give the formal definition. With respect to notation, we place a ∼ above those symbols that denote constituents of data-aware CMS (but we use the same symbols as for CMS without constraints).

**Definition 19** (Constraint coloring). A constraint coloring $\bar{c}$ over $[N \subseteq \text{Node}, DC \subseteq \mathbb{D} \text{C}(N)]$ is a pair $\bar{c} = \langle c, dc \rangle$ with $c$ a coloring over $N$ and $dc \in DC$ a data constraint such that $dc \in \mathbb{D} \text{C}(F)$ with $F = \{ n \in N \mid c(n) = \text{no-flow color} \}$. We denote the set of all constraint colorings over $[N, DC]$ by $\text{CCol}(N, DC)$.

Note that our definition does not exclude a constraint coloring $\bar{c} = \langle c, dc \rangle$ with inconsistent $c$ and $dc$. For example, $c$ may mark some node $n$ with the no-flow color, while $dc$ entails the flow of some data item through $n$. We do
not forbid such constraint colorings, because they do not impair the models in which they appear: they merely describe behavior that never arises.

Next, we incorporate data constraints in the definitions of the other constituents of ordinary CMs (as presented in Section 3.3). This turns out straightforwardly. To summarize the upcoming definitions: (i) a constraint coloring table is a set of constraint colorings, (ii) a constraint ctm is a map from indexes to constraint coloring tables, (iii) a constraint next function is a map from [index, constraint coloring]-pairs to indexes, and (iv) a constraint cm (CCM) is a [constraint next function, index]-pair, i.e., a data-aware cm.

**Definition 20** (Constraint coloring table). A constraint coloring table \( \widetilde{T} \) over \([N \subseteq \text{Node}, DC \subseteq \text{Dc}(N)]\) is a set \( \widetilde{T} \subseteq \text{CCol}(N, DC) \) of constraint colorings over \([N, DC]\).

**Definition 21** (Constraint ctm). A constraint ctm \( \widetilde{S} \) over \([N \subseteq \text{Node}, DC \subseteq \text{Dc}(N), I \subseteq \text{Index}]\) is a function \( \widetilde{S} : I \rightarrow \wp(\text{CCol}(N, DC)) \) that maps an index \( i \) to a constraint coloring table \( \widetilde{S}(i) \) over \([N, DC]\).

**Definition 22** (Constraint next function). Let \( \widetilde{S} \) be a constraint ctm over \([N, DC, I]\). A constraint next function \( \widetilde{\eta} \) over \( \widetilde{S} \) is a partial function \( \widetilde{\eta} : I \times \text{CCol}(N, DC) \rightarrow I \) that maps every [index, constraint coloring]-pair \( (i, \widetilde{c}) \) such that \( \widetilde{c} \in \widetilde{S}(i) \) to an index \( \widetilde{\eta}(i, \widetilde{c}). \)

**Definition 23** (ccm). Let \( \widetilde{S} \) be a constraint ctm over \([N, DC, I]\). A ccm \( \mathcal{CM} \) over \( \widetilde{S} \) is a pair \( \mathcal{CM} = (\widetilde{\eta}, i_0) \) with \( \widetilde{\eta} \) a constraint next function over \( \widetilde{S} \) and \( i_0 \in I \).

To illustrate the previous definitions, Figure 8 shows the constraint ctms of LossySync and FIFO for \( \text{Data} = \{\text{"foo"}\} \).
execution step of LossySync wherein the same data item flows through \( n_1 \) and \( n_2 \). Constraint coloring \((c_1, \top)\), describes the “execution step” wherein a connector idles. Due to the constraint \( \top \), this may always happen. Similarly, LossySync can always behave as described by \((c_2, \top)\), but whereas \((c_4, \top)\) entails no flow at all, \((c_2, \top)\) entails flow through \( c_1 \) and no flow through \( c_2 \). Here, \( \top \) specifies that we do not care about which data item flows through \( c_1 \).

With respect to FIFO, for simplicity of the example, we assume the universe of data items a singleton as before (see Section 3.3).

Finally, we update the join operators for CMS to incorporate data constraints. For brevity, we give these definitions only for constraint colorings and constraint coloring tables. The join operators for constraint CTMs (symbol: \( \bowtie \)), constraint next functions (symbol: \( \ominus \)), and CCMS (symbol: \( \bowtie \ominus \)) resemble their respective join operators in Section 3.3: essentially, it suffices to replace \( S_1, S_2, \eta_1, \eta_2, CM_1, \) and \( CM_2 \) in Definitions 16–18 with their \( \sim \) versions \( S_1, S_2, \eta_1, \eta_2, CM_1, \) and \( CM_2 \). We require only these minor updates, because our extension of CMS with data constraints affects only the definition of colorings directly. For completeness, in Appendix A, we give the definitions of the remaining join operators.

**Definition 24** (Join of constraint colorings). Let \( \overline{c}_1 = (c_1, dc_1) \) and \( \overline{c}_2 = (c_2, dc_2) \) be constraint colorings over \([N_1, DC_1]\) and \([N_2, DC_2]\) such that \( c_1(n) = c_2(n) \) for all \( n \in N_1 \cap N_2 \). Their join, denoted by \( \overline{c}_1 \cup \overline{c}_2 \), is a constraint coloring over \([N_1 \cup N_2, DC_1 \land DC_2]\) defined as:

\[
\overline{c}_1 \cup \overline{c}_2 = (c_1 \cup c_2, dc_1 \land dc_2).
\]

**Definition 25** (Join of constraint coloring tables). Let \( \bar{T}_1 \) and \( \bar{T}_2 \) be constraint coloring tables over \([N_1, DC_1]\) and \([N_2, DC_2]\). Their join, denoted by \( \bar{T}_1 \odot \bar{T}_2 \), is a constraint coloring table over \([N_1 \cup N_2, DC_1 \land DC_2]\) defined as:

\[
\bar{T}_1 \odot \bar{T}_2 = \left\{ \bar{c}_1 \cup \bar{c}_2 \mid \bar{c}_1 = (c_1, dc_1) \in \bar{T}_1 \text{ and } \bar{c}_2 = (c_2, dc_2) \in \bar{T}_2 \right\}.
\]

Note that by taking their conjunction, we combine data constraints in Definition 24 in the same way as the join operator for CA in Definition 7. The lemmas that we formulate and prove in the subsequent sections establish the appropriateness of taking the conjunction of data constraints in the context of CMS.

To illustrate the previous definitions, Figure 9 shows the constraint CTM of LossyFIFO. The colorings \( c_i \) with \( i \in \{12, 24, 43, 44\} \) refer to the colorings in Figure 6.
We remark that in [61], Proença uses pairs of colorings and records (as in Definition 3) as transition labels of his behavioral automata (see Section 3.2.3) to account for the transfer of data that takes place through data-flows described by those colorings. This suggests using [coloring, record]-pairs as a data-aware cm. However, our constraint-based extension offers a more concise formalization. For instance, in Proença’s model, to give the semantics of LossySync, one must include a [coloring, record]-pair in its coloring table for each data item in Data. With our constraint-based extension, in contrast, we capture this with a single constraint-based coloring as shown in Figure 8.

5 From CCMs to CA

In this section, we present a unary operator, denoted by $L$, which takes a CCM as its input and outputs an equivalent CA; shortly, we elaborate on the meaning of “equivalence” in this context. We call this process of transforming a CCM to a CA the $L$-transformation. By defining the $L$-transformation for any CCM, it follows that the class of connectors that we can model with a CA includes those that we can model with a CCM. It follows that CA are at least as expressive as data-aware CCMs.

Suppose we wish to transform a CCM $\tilde{C}M = \langle \tilde{\eta}, i_0 \rangle$ over $\tilde{S}$ over $[N, DC, I]$. The $L$-operator derives a CA from $\tilde{C}M$ as follows. First, $L$ instantiates the set of states of this derived CA with the set of indexes $I$: this seems reasonable as $I$ denotes the set of indexes that represent the states of the connector that $\tilde{C}M$ models. Next, $L$ constructs a transition relation $T$ based on the mappings in $\tilde{\eta}$: for each $\langle i, \langle c, dc \rangle \mapsto i' \rangle \in \tilde{\eta}$, the $L$-operator creates a transition from state $i$ to state $i'$, labeled with $dc$ as its data constraint and with the set of nodes to which $c$ assigns the flow color as its firing set. Finally, $i_0$ becomes the initial state of the new CA.
Definition 26 (L). Let $\widehat{\mathcal{C}M} = \langle \widehat{\eta}, i_0 \rangle$ be a CCM over $\widehat{S}$ over $[N, DC, I]$. The $L$-transformation of $\widehat{\mathcal{C}M}$, denoted by $L(\widehat{\mathcal{C}M})$, is defined as:

$$L(\widehat{\mathcal{C}M}) = \langle I, T, i_0 \rangle$$

with:

$$T = \left\{ \langle i, F, dc, \widehat{\eta}(i, \widehat{c}) \rangle \mid i \in I \text{ and } \widehat{c} = \langle c, dc \rangle \in \widehat{S}(i) \text{ and } F = \{ n \in N \mid c(n) = \_ \} \right\}.$$  

The proposition below states that the application of $L$ to a CCM yields a CA.

Proposition 1. Let $\widehat{\mathcal{C}M}$ be a CCM over $\widehat{S}$ over $[N, DC, I]$. Then, $L(\widehat{\mathcal{C}M})$ is a CA over $[N, DC]$.

Proof. See Appendix B. $\square$

In the rest of this section, we prove the equivalence between a CCM $\widehat{\mathcal{C}M}$ and the CA that results from applying $L$ to $\widehat{\mathcal{C}M}$. Additionally, we prove the compositionality of $L$.

5.1 Correctness of $L$

In this subsection, we prove the correctness of $L$: we consider $L$ correct if its application to a CCM yields an equivalent CA. We call a CCM and a CA equivalent if there exists a bisimulation relation that relates these two models. Informally, a CA $\mathcal{C}A$ is bisimilar to a CCM $\widehat{\mathcal{C}M}$ if, for each mapping in the constraint next function of $\widehat{\mathcal{C}M}$, there exists a corresponding transition in the CA and vice versa, i.e., a transition that describes the same behavior in terms of the nodes that fire, the data items that flow, and the change of state.

Definition 27 (Bisimulation). Let $\mathcal{C}A = \langle Q, T, q_0 \rangle$ be a CA over $[N, DC]$ and $\widehat{\mathcal{C}M} = \langle \widehat{\eta}, i_0 \rangle$ a CCM over $\widehat{S}$ over $[N, DC, I]$. $\mathcal{C}A$ and $\widehat{\mathcal{C}M}$ are bisimilar, denoted as $\mathcal{C}A \sim \widehat{\mathcal{C}M}$, if there exists a relation $R \subseteq Q \times I$ such that $\langle q_0, i_0 \rangle \in R$ and for all $\langle q, i \rangle \in R$:

(1) If $\langle q, F, dc, q' \rangle \in T$ then there exists an $i' \in I$ such that:

- $[(i, \widehat{c}) \mapsto i'] \in \widehat{\eta}$ with $\widehat{c} = \langle c, dc \rangle$;
- $\langle q', i' \rangle \in R$;
- $F = \{ n \in N \mid c(n) = \_ \}$.

(2) If $[(i, \widehat{c}) \mapsto i'] \in \widehat{\eta}$ with $\widehat{c} = \langle c, dc \rangle$ then there exists a $q' \in Q$ such that:

- $\langle q, F, dc, q' \rangle \in T$;
- $\langle q', i' \rangle \in R$;
- $F = \{ n \in N \mid c(n) = \_ \}.$

In that case, $R$ is called a bisimulation relation.
In the previous definition, we compare data constraints syntactically. Alternatively, one can define bisimulation by comparing data constraints logically, i.e., in terms of the records that satisfy a data constraint. The latter yields a weaker notion of bisimulation than the one formalized in Definition 27. Because we can prove the stronger form between a CCM $\tilde{CM}$ and its $L$-transformation $L(\tilde{CM})$, however, we compare data constraints syntactically. In our proof, which appears in the appendix, we choose the diagonal relation on the set of indexes as a bisimulation relation.

**Lemma 1.** Let $\tilde{CM}$ be a CCM. Then, $L(\tilde{CM}) \sim \tilde{CM}$.

*Proof.* See Appendix B. \qed

### 5.2 Compositionality of $L$

We end this section with a compositionality lemma for $L$. Informally, it states that it does not matter whether we first join CCMs $\tilde{CM}_1$ and $\tilde{CM}_2$ and then apply $L$ to the resulting composite or first apply $L$ to $\tilde{CM}_1$ and $\tilde{CM}_2$ individually and then join the resulting transformations; the resulting CA equal each other. The relevance of this result lies in the potential reduction in the amount of overhead that it allows for when applying the $L$-operator in practice. This works as follows. There exist tools for Reo that operate on CA and that have built-in functionality for the computation of their join. By the compositionality lemma for $L$, to use such a tool, we need to transform the common primitives only once, store these in a library, and use this library together with the built-in functionality for join computation to construct the CA of composites (on which the tool subsequently operates). Thus, the overhead of this approach remains constant. In contrast, the overhead of the alternative—first joining CCMs and then transforming the resulting composites using $L$—grows linearly in the number of composites one wishes to apply the tool on.\footnote{For now, we assume the computation of the join of two CCMs and two CA equally expensive. In Section 7, we argue for the merits of our approach when the cost of joining CCMs differs from the cost of joining CA.} In Section 7, we illustrate the foregoing with a concrete example; here, we proceed with the lemma.

**Lemma 2.** Let $\tilde{CM}_1$ and $\tilde{CM}_2$ be CCMs. Then, $L(\tilde{CM}_1) \bowtie L(\tilde{CM}_2) = L(\tilde{CM}_1 \bowtie \tilde{CM}_2)$.

*Proof.* See Appendix B. \qed
Although we consider only CCMs with two colors, one can apply \( L \) also to CCMs with three colors. In fact, Lemma 1 (correctness) would still hold! Essentially, this means that CCMs with three colors do not have a higher degree of expressiveness than CCMs with two colors. In contrast, Lemma 2 (compositionality) does not hold if we consider CCMs with three colors. See Appendix B for details.

6 From CA to CCMs

In this section, we demonstrate a correspondence between CCMs and CA in the direction opposite to the previous section's: from the latter to the former. Our approach, however, resembles our approach in Section 5: we present a unary operator, denoted by \( \mathbb{L} \), which takes a CA as its input and produces an equivalent, i.e., bisimilar, CCM.\(^{27}\) We call our process of transforming a CA to a CCM the \( \mathbb{L} \)-transformation and define the \( \mathbb{L} \)-operator for any CA. It follows that the class of connectors that we can model with CA includes those that we can model with a CCM. Since the previous section gave us a similar result in the opposite direction, we conclude that CCMs and CA have the same degree of expressiveness.

The \( \mathbb{L} \) operator works as follows; suppose we wish to transform a CA \( \text{CA} \) over \([N, DC]\). The \( \mathbb{L} \)-operator derives a CCM from \( \text{CA} \) as follows: for each transition \( \langle q, F, dc, q' \rangle \) in the transition relation of \( \text{CA} \), \( \mathbb{L} \) includes a mapping from state \( q \) and a constraint coloring \( \bar{c} = \langle c, dc \rangle \) to state \( q' \), where \( c \) assigns the flow color to all and only the nodes in \( F \).

**Definition 28 \((\text{col})\).** Let \( N, F \subseteq \text{NODE} \). Then:

\[
\text{col}(N, F) = \left\{ n \mapsto \kappa \mid n \in N \text{ and } \kappa = \begin{cases} \text{flow color} & \text{if } n \in F \\ \text{otherwise} \end{cases} \right\}.
\]

**Definition 29 \((\mathbb{L})\).** Let \( \text{CA} = \langle Q, T, q_0 \rangle \) be a CA over \([N, DC]\). The \( \mathbb{L} \)-transformation of \( \text{CA} \), denoted by \( \mathbb{L}(\text{CA}) \), is defined as:

\[
\mathbb{L}(\text{CA}) = \langle \bar{n}, q_0 \rangle
\]

\(^{27}\)Recall from Section 3.2.1 (and Footnotes 8 and 9) that, without loss of generality, we consider only CA whose transition relations satisfy the following condition:

\[
\langle q, F_1, dc_1, q'_1 \rangle, \langle q, F_2, dc_2, q'_2 \rangle \in T \text{ and } q'_1 \neq q'_2 \text{ implies } \langle F_1, dc_1 \rangle \neq \langle F_2, dc_2 \rangle
\]

This condition ensures that we can cast transition relations into transition functions.
with: \( \bar{\eta} = \{ (q, (\text{co1}(N,F),dc)) \mapsto q' | (q,F,dc,q') \in T \} \).

The proposition below states that the application of \( \frac{1}{L} \) to a CA yields a ccm.

**Proposition 2.** Let \( \text{CA} = (Q, T, q_0) \) be a CA over \( [N,DC] \). Then, \( \frac{1}{L}(\text{CA}) \) is a ccm over \( \bar{S} \) over \( [N,DC,Q] \) defined as:

\[
\bar{S} = \{ q \mapsto \bar{T} | q \in Q \text{ and } \bar{T} = \{ (\text{co1}(N,F),dc) | (q,F,dc,q') \in T \} \}.
\]

**Proof.** See Appendix B.

\[ \square \]

### 6.1 Inverse

Having defined \( \frac{1}{L} \), we proceed by proving that it forms the inverse of \( L \) (as already hinted at by its symbol) and vice versa. We do this before stating the correctness and compositionality of \( \frac{1}{L} \), because the proofs of these lemmas become significantly easier once we know that \( \frac{1}{L} \) inverts \( L \). The following two lemmas state the inverse properties in both directions.

**Lemma 3.** Let \( \bar{\text{CM}} \) be a ccm. Then, \( \frac{1}{L}(L(\bar{\text{CM}})) = \bar{\text{CM}} \).

**Proof.** See Appendix B.

\[ \square \]

**Lemma 4.** Let \( \text{CA} \) be a CA. Then, \( L(\frac{1}{L}(\text{CA})) = \text{CA} \).

**Proof.** See Appendix B.

\[ \square \]

### 6.2 Correctness and Compositionality of \( \frac{1}{L} \)

As mentioned previously, knowing that \( L(\frac{1}{L}(\text{CA})) = \text{CA} \) simplifies our correctness and compositionality proofs. We start with the former. Lemma 5, which appears below, states the bisimilarity between \( \text{CA} \) and its \( \frac{1}{L} \)-transformation \( \frac{1}{L}(\text{CA}) \).

**Lemma 5.** Let \( \text{CA} \) be a CA. Then, \( \text{CA} \sim \frac{1}{L}(\text{CA}) \).

**Proof.** See Appendix B.

\[ \square \]

Finally, Lemma 6 states the compositionality of \( \frac{1}{L} \): informally, this means that it does not matter whether we first join \( \text{CA} \) \( \text{CA}_1 \) and \( \text{CA}_2 \) and then apply \( \frac{1}{L} \) to the resulting composite or first apply \( \frac{1}{L} \) to \( \text{CA}_1 \) and \( \text{CA}_2 \) and then join the resulting transformations; the resulting ccms equal each other. Our proof, similar to that of the previous lemma, relies on the inverse lemmas.
7 Application

In this section, we sketch an application of the results presented in Sections 5 and 6: the integration of verification and animation of context-sensitive connectors in Vereofy [16], a model checking tool that operates on CA.28 Broadly, this application consists of two parts: model checking connectors built from context-sensitive constituents and generating animated counterexamples.

Verification of CCMS Vereofy operates on CA, and therefore, many consider it unable to verify context-sensitive connectors. We mend this deficiency as follows. First, recall from Section 3.3.1 that we can transform CMSs with three colors, known for their ability to properly capture context-sensitivity to corresponding CMSs with two colors. Because data constraints form an orthogonal concern, this means that CMSs with only two colors can serve as faithful models of context-sensitive connectors. Consequently, the results from Section 5 enable the verification of such connectors with Vereofy: using the L-transformation, we transform context-sensitive CMSs to context-sensitive CA, which we then can analyze with Vereofy.

As an example, Figure 10 shows the CA obtained by applying L to the encoded 3CMSs, as 2CMSs, of LossySync and FIFO for DATA = {“foo”}.

Lemma 6. Let CA1 and CA2 be CA. Then, $\frac{1}{E}(CA_1) \gtrless \frac{1}{E}(CA_2) = \frac{1}{E}(CA_1 \Leftrightarrow CA_2)$.

Proof. See Appendix B.

28http://www.vereofy.de
of concrete nodes in the CA represents a single conceptual node that one would draw in a diagram. Note that abstracting away \( \bar{n}_1, \bar{n}_2, \) and \( \bar{n}_3 \), yields the CA in Figure 3. The left CA in Figure 11 shows the join of the CA in Figure 10; the right CA in Figure 11 shows the same CA but with \( \bar{n}_1, \bar{n}_2, \) and \( \bar{n}_3 \) abstracted away. Compared to Figure 4, this CA does not contain the transition \( \langle \text{LFIFO-E, } \{n_1\}, \top, \text{LFIFO-E} \rangle \), i.e., it models the context-sensitive LossySync rather than its nondeterministic sibling. One can now use Vereofy to analyze this CA. Another example appears in [42].

In this application, the compositionality of \( \mathcal{L} \) in Lemma 2 plays an important role (as already outlined in Section 5.2): it facilitates the one-time-application of \( \mathcal{L} \) to encoded \( \text{CCMs} \), as \( \text{CCMs} \), of Reo’s primitives. Subsequently, one can use Vereofy’s built-in functionality for joining the compound automata that one wishes to analyze. We remark that our compositionality lemmas work also in the opposite direction: if future studies indicate that joining \( \text{CCMs} \) costs less than joining \( \text{CAs} \), we can extend Vereofy with a module to automatically (i) transform CA of primitives to \( \text{CCMs} \) with \( \mathcal{L} \), (ii) join the resulting \( \text{CCMs} \), and (iii) transform the resulting composite back to a CA with \( \mathcal{L} \). (To truly gain in performance, however, the costs of transforming forth and back should not exceed the benefits of joining \( \text{CCMs} \) instead of \( \text{CAs} \).)

**Animation of CA** Vereofy facilitates the generation and inspection of counterexamples, an important feature that distinguishes it from mCRL2 (recall from Section 3.4 that, alternatively, we can verify Reo connectors with the analysis tools of mCRL2). When used together with the Ect, Vereofy can in some cases display counterexamples as connector animations. These animated counterexamples comprise a graphical model of a connector through
which data items visually flow for each execution step in an error trace.

Although such animations improve the ease with which Vereofy users can analyze counterexamples, the opportunity to actually provide these visualizations depends on the availability of a CM of the connector under investigation (in addition to the CA that Vereofy’s verification algorithm operates on). Moreover, the standalone version of Vereofy, a command-line tool, does not facilitate the animation of counterexamples at all. The results in Section 6, however, enable animated counterexamples for any CA: in case of unavailability of a CA, Vereofy can simply generate such a model with the $\frac{1}{L}$-transformation.

8 Conclusion

In this paper, we gave an overview of all the existing semantic formalisms for modeling Reo connectors. Furthermore, we showed that once extended with data constraints, coloring models with two colors and constraint automata have the same degree of expressiveness by defining two operators that transform such data-aware CMs to corresponding CA and vice versa. Moreover, we showed the compositionality of these operators. Though primarily a theoretical contribution, we illustrated how our results can broaden the applicability of Reo’s tools.

With respect to future work, we would like to implement the transformation operators and the sketched extension to Vereofy. Another application worth investigating comprises the development of an implementation of Reo based on transforming behavioral models of connectors back and forth to improve performance. Finally, we would like to study correspondences between other classes of semantic models.

Acknowledgements

We thank the reviewers and the members of the ICE 2011 forum nicknamed gege, wind, wolf and xyz for their valuable comments.

References


A Appendix: Composition Operators

In this appendix, we give the formal definitions of the join operators whose definition we gave only informally in Section 4. More specifically, we give the definitions of the join operators for constraint CTMs, constraint next functions, and CCMs. The definitions of the join operators for constraint colorings and constraint coloring tables appear in Section 4. As mentioned in that section, we obtain the operators that we define below by replacing $S_1, S_2, \eta_1, \eta_2, CM_1,$ and $CM_2$ in Definitions 16–18 with their $\sim$ versions.

**Definition 30** (Join of constraint CTMs). Let $\tilde{S}_1$ and $\tilde{S}_2$ be constraint CTMs over $[N_1, DC_1, I_1]$ and $[N_2, DC_2, I_2]$. Their join, denoted by $\tilde{S}_1 \odot \tilde{S}_2$, is a constraint CTM over $[N_1 \cup N_2, DC_1 \land DC_2, I_1 \times I_2]$ defined as:

$$\tilde{S}_1 \odot \tilde{S}_2 = \{\langle i_1, i_2 \rangle \mapsto \tilde{S}_1(i_1) \cdot \tilde{S}_2(i_2) | i_1 \in I_1 \text{ and } i_2 \in I_2\}.$$ 

**Definition 31** (Join of constraint next functions). Let $\tilde{\eta}_1$ and $\tilde{\eta}_2$ be constraint next functions over $\tilde{S}_1$ and $\tilde{S}_2$ over $[N_1, DC_1, I_1]$ and $[N_2, DC_2, I_2]$. Their join, denoted by $\tilde{\eta}_1 \odot \tilde{\eta}_2$, is a constraint next function over $\tilde{S}_1 \odot \tilde{S}_2$ defined as:

$$\tilde{\eta}_1 \odot \tilde{\eta}_2 = \left\{ \langle i_1, i_2 \rangle, \tilde{c}_1 \cup \tilde{c}_2 \mid \langle i_1, i_2 \rangle \in I_1 \times I_2 \text{ and } \langle \tilde{\eta}_1(i_1, \tilde{c}_1), \tilde{\eta}_2(i_2, \tilde{c}_2) \rangle \in (\tilde{S}_1 \odot \tilde{S}_2)((i_1, i_2)) \right\}.$$ 

**Definition 32** (Join of CCMs). Let $\tilde{CM}_1 = \langle \tilde{\eta}_1, i_0^1 \rangle$ and $\tilde{CM}_2 = \langle \tilde{\eta}_2, i_0^2 \rangle$ be CCMs over $\tilde{S}_1$ and $\tilde{S}_2$. Their join, denoted by $\tilde{CM}_1 \odot \tilde{CM}_2$, is a CCM over $\tilde{S}_1 \odot \tilde{S}_2$ defined as:

$$\tilde{CM}_1 \odot \tilde{CM}_2 = \langle \tilde{\eta}_1 \odot \tilde{\eta}_2, (i_0^1, i_0^2) \rangle.$$
B  Appendix:  Proofs

Proof of Proposition 1 (\(L\) is well-defined). By Definition 26 of \(L\), we must show that \(L(\mathcal{M}) = \langle I, T, i_0 \rangle\) is a CA over \([N, DC]\). To demonstrate this, by Definition 5 of CA, we must show that \(T \subseteq I \times \wp(N) \times DC \times I\). Now, suppose \(\mathcal{M} = \langle \eta, i_0 \rangle\). Because, by the premise, \(\mathcal{M}\) is a ccm over \(\tilde{S}\) over \([N, DC, I]\), by Definition 22, the co-domain of \(\eta\) is \(I\). Also, by Definition 26, for all \(\langle i, F, dc, \eta(i, \tilde{c}) \rangle \in T\) with \(\tilde{c} = \langle c, dc \rangle\), it holds that \(i \in I, F \subseteq N, \) and \(dc \in DC\) (this latter follows from Definition 19).

Proof of Lemma 1 (correctness of \(L\)). Suppose \(\tilde{CM} = \langle \tilde{\eta}, i_0 \rangle\) over \([N, \tilde{S}]\) over \([N, DC, I]\). Additionally, suppose \(L(\mathcal{M}) = \langle I, T, i_0 \rangle\). We show that \(\mathcal{R} = \{\langle i, i \rangle \mid i \in I\}\) is a bisimulation relation by demonstrating that it satisfies (i) and (ii) in Definition 27. Let \(\langle i, i \rangle \in \mathcal{R}\).

(i) Suppose \(\langle i, F, dc, i' \rangle \in T\). Then, by Definition 26 of \(L\), there exists a \(\tilde{c} = \langle c, dc \rangle \in \tilde{S}(i)\) such that \(i' = \tilde{\eta}(i, \tilde{c})\). Hence, \(\langle i, \tilde{c} \rangle \mapsto i' \in \eta\). Also, by the definition of \(\mathcal{R}\), \(\langle i', i' \rangle \in \mathcal{R}\). Finally, by Definition 26 of \(L\), \(F = \{n \in N \mid c(n) = \ldots\}\). Therefore, \(\mathcal{R}\) satisfies (i).

(ii) Suppose \(\langle i, \tilde{c} \rangle \mapsto i' \in \eta\) with \(\tilde{c} = \langle c, dc \rangle\). Then, by Definition 22 of \(\tilde{\eta}\), it holds that \(i \in I\) and \(\tilde{c} \in \tilde{S}(i)\), hence by Definition 26 of \(L\), \(\langle i, F, dc, i' \rangle \in T\) with \(F = \{n \in N \mid c(n) = \ldots\}\). Also, by the definition of \(\mathcal{R}\), \(\langle i', i' \rangle \in \mathcal{R}\). Therefore, \(\mathcal{R}\) satisfies (ii).

Thus, \(\mathcal{R}\) satisfies (i) and (ii). Moreover, because \(i_0 \in I\) by Definition 23, \(\langle i_0, i_0 \rangle \in \mathcal{R}\). Hence, \(\mathcal{R}\) is a bisimulation relation. Therefore, \(L(\mathcal{M}) \sim \mathcal{M}\).  

Proof of Lemma 2 (compositionality of \(L\)). Suppose \(\mathcal{M}_1 = \langle \eta_1, i_{01} \rangle\) and \(\mathcal{M}_2 = \langle \eta_2, i_{02} \rangle\) over \(\tilde{S}_1\) and \(\tilde{S}_2\). Applying Definition 26 of \(L\) to the left-hand side (LHS) and Definition 32 of \(\tilde{\triangle}\) (informally on page 231) to the right-hand side (RHS) yields:
What remains to be shown is that
Applying Definition 7 of \( \triangleleft \triangleleft \)
\( T \) with:
\[
\begin{align*}
    T_1 &= \left\{ \langle i_1, F_1, dc_1, \eta_1(i_1, c_1) \rangle \mid i_1 \in I_1, dc_1 \in \eta_1(i_1) \quad \text{and} \quad F_1 = \{ n \in N_1 \mid c_1(n) = 1 \} \right\} \\
    T_2 &= \left\{ \langle i_2, F_2, dc_2, \eta_2(i_2, c_2) \rangle \mid i_2 \in I_2, dc_2 \in \eta_2(i_2) \quad \text{and} \quad F_2 = \{ n \in N_2 \mid c_2(n) = 1 \} \right\}
\end{align*}
\]
Applying Definition 7 of \( \triangleright \) (to LHS), and Definition 26 of \( L \) (to RHS) yields:
\[
\langle I_1 \times I_2, T, \langle i_{10}^1, i_{10}^2 \rangle \rangle = \langle I_1 \times I_2, T', \langle i_{10}^1, i_{10}^2 \rangle \rangle
\]
with:
\[
T = \left\{ \langle \langle i_1, i_2 \rangle, F_1 \cup F_2, dc_1 \land dc_2, \langle i'_{1}, i'_2 \rangle \rangle \mid \langle i_1, F_1, dc_1, i'_1 \rangle \in T_1 \quad \text{and} \quad \langle i_2, F_2, dc_2, i'_2 \rangle \in T_2 \quad \text{and} \quad F_1 \cap N_2 = F_2 \cap N_1 \right\}
\]
and:
\[
T' = \left\{ \langle i, F, dc, (\eta_1 \circ \eta_2)(i, c) \rangle \mid i \in I_1 \times I_2, dc \in F_1 \cap N_2 = F_2 \cap N_1 \right\}
\]
What remains to be shown is \( T = T' \). This follows from Figure 12. \( \square \)

Interestingly, Lemma 2 (compositionality) does not hold if we consider CCMs with three colors: the fourth—counted from top to bottom—equality in Figure 12 on page 249 ("Because, by the definition of \( F_1 \) and \( F_2 \)...") becomes invalid if we consider CCMs with three colors. This means that, although CCMs with two and three colors have the same degree of expressiveness, they compose differently: paradoxically, the addition of a third color restricts, as intended, the number of compatible colorings. This restriction allows one to model context-sensitive connectors compositionally with three colors: it serves as a filter for nondeterministic behavior that should not occur due to context-sensitivity.

Proof of Proposition 2 (\( L \) is well-defined). By Definition 29 of \( L \), we must show that \( L(CA) = \langle \eta_1, q_0 \rangle \) is a CCM over \( \tilde{S} \). To demonstrate this, by Definition 23 of CCM, we must show that \( \tilde{\eta} \) is a constraint next function over \( \tilde{S} \). First, by Definition 28, all colorings that occur in the domain of \( \tilde{\eta} \) have \( N \) as their domain. Next, by Definition 29, all indexes that occur in the domain
Figure 12: Proof: \( T = T' \).

and co-domain of \( \tilde{\eta} \) are states that appear in elements of the transition relation \( T \); therefore, by Definition 5, all indexes come from \( Q \). Similarly, by Definition 29, all data constraints that occur in the domain of \( \tilde{\eta} \) also appear in elements of \( T \), hence come from DC. Finally, by Definition 22, we must show that \( \tilde{\eta} \) maps every \((i, \tilde{c})\) such that \( \tilde{c} \in S(i) \) to an index in \( Q \). This follows from Definition 29 and the definition of \( \tilde{S} \) in the premise.

Proof of Lemma 3 (left-inverse for \( L \)). Suppose \( \widetilde{CM} = (\tilde{\eta}, i_0) \) is defined over \( \tilde{S} \) over \([N, DC, I] \). Then, \( \frac{1}{L}(L(\widetilde{CM})) = \widetilde{CM} \) follows from Figure 13.
\[ L(CM) = CM. \]

**Proof of Lemma 4 (right-inverse for \( L \)).** Suppose \( CA = (Q, T, q_0) \) is defined over \( [N, DC] \). Then, \( L(1_L(CA)) = CA \) follows from Figure 14. □

**Proof of Lemma 5 (\( 1_L \) is correct).** Suppose \( CM = 1_L(CA) \). Then, \( CA \sim 1_L(CA) \) iff \( L(1_L(CA)) \sim L(CM) \) iff \( L(CM) \sim CM \) by Lemma 4. The latter follows from Lemma 1. □

**Proof of Lemma 6 (compositionality of \( 1_L \)).** Follows from Figure 15. □
\[ L(\underline{\underline{CA}}) = CA \]

**Figure 14:** Proof: \( L(\underline{\underline{CA}}) = CA \).

\[ \underline{\underline{CA}} \triangleleft \underline{\underline{CA}} \]

**Figure 15:** Proof: \( \underline{\underline{CA}} \triangleleft \underline{\underline{CA}} = \underline{\underline{CA}} \triangleleft \underline{\underline{CA}} \).

© Scientific Annals of Computer Science 2012