

A Constraint-based Method to Compute Semantics of Channel-based Coordination Models

Behnaz Changizi, Natallia Kokash
Leiden Institute of Advanced Computer Science (LIACS)
Leiden, The Netherlands
b.changizi@umail.leidenuniv.nl, nkokash@liacs.nl

Farhad Arbab
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
farhad.arbab@cwi.nl

Abstract—Reo is an exogenous channel-based coordination language that acts as glue code to tie together software components and services. The building blocks of Reo models are connectors that impose constraints on the data-flow in component or service-based architectures in terms of data synchronization, buffering, mutual exclusion, etc. Several semantic models have been introduced to formalize the behavior of Reo. These models differ in terms of expressiveness, computation complexity and purposes that they serve. In this paper, we present a method and a tool for building formal automata-based semantics of Reo that unifies various aspects of existing semantics. We express the behavior of a Reo network as a mixed system of Boolean and numerical constraints constructed compositionally by conjuncting the assertions for its constituent parts. The solutions of this system are found with the help of off-the-shelf constraint solvers and are used to construct the constraint automaton with state memory that gives the sound and complete semantics of Reo with respect to existing models. Our approach is more efficient compared to the existing methods for generating formal semantics of Reo connectors.

Keywords—formal semantics; Reo; constraint automata; coloring semantics; constraint solving.

I. INTRODUCTION

Service-oriented architecture [1] (SOA) is an indispensable solution for many of today's problems. The SOA implementation depends on a mesh of functionality units, called services. Services are loosely coupled and do not invoke or communicate with each other directly. Instead, they employ a pre-defined protocol, which specifies the way they can exchange messages amongst themselves. As a result, the correctness of a SOA implementation relies not only on the correctness of its involved services but also on the properness of its communication protocol.

Coordination languages and models provide dedicated frameworks to study the communication protocols as separate concerns. They define the “glue code” that ties together the services to enable the message passing among the involved services. Some recent coordination models include: i) a Calculus for Orchestration of Web Services (COWS) [2], which specifies the combination of service-oriented applications and models their dynamic behavior; ii) Orc [3], a process calculus for distributed and concurrent programming which provides uniform access to computational services,

including distributed communication and data manipulation; and iii) Reo [4], an exogenous coordination language that realizes the coordination patterns in terms of its complex connectors, also called *networks*, that are built out of simple primitives called *channels*. In the sequel, we focus on Reo.

Each channel in Reo defines a form of coordination in terms of synchronizing, buffering, retaining data, etc., along with constraining its input and output data items. Reo allows hierarchical modeling where arbitrarily complex connectors can be formed out of simpler networks. In our previous work [5] [6], we have presented the suitability of Reo to model behavioral patterns describable by business process models. We have also developed tools for automatic transformation of these models into Reo [6]. This enables the use of Reo analysis methods and tools on the coordination protocols that originally were not expressed in Reo.








To perform formal analysis on Reo networks, formal semantics of these models are necessary. Several operational semantics have been proposed for Reo [7] with various styles of I/O streams [4], automata, coloring [8] and constraints [9]. The most basic automata-based semantics of Reo is Constraint Automata (CA) [10]. An advantage of CA and its extensions is their support for data-constraints that are part of the coordination primitives in Reo. This is in contrast with the coloring semantics that abstracts data-flow and expresses the behavior of a connector only in terms of existence or lack of data-flow.

Constraint Automata with State Memory (CASMs) [11] is an extension of CA that due to its state abstraction and data-awareness, is suitable as a more compact semantic model for Reo in model checking. In this paper, we present a constraint-based technique and a tool to generate CASMs from Reo networks in a compositional manner.

A shortcoming of earlier work stems from its lack of support for data-dependent behavior. We overcome this shortcoming in the work we present in this paper. Our tool is a necessary step for providing fully automated model checking for data-aware and context-dependent composition of services coordinated by Reo.

The rest of this paper is organized as follows. In Section 2, we explain the basics of Reo. In Section 3, we introduce

Table I: Graphical Representation of the Most Frequently Used Reo Channels

	A <i>Sync</i> channel accepts data from its source end iff it can dispense it simultaneously through its sink end.
	A <i>LossySync</i> reads a data item from its source end and writes it simultaneously to its sink end. If the sink end is not ready to accept the data item, the channel loses it.
	A <i>SyncDrain</i> reads data to discard through its two source ends iff both ends are ready to interact simultaneously.
	An <i>AsyncDrain</i> accepts and discards a data item from either of its source ends that offers one. If both ends offer data items simultaneously, then the channel chooses one non-deterministically.
	A <i>Filter</i> accepts a data item that does not match its predefined filter pattern P from its source and loses it. For a data item that matches its filter pattern P , a filter channel behaves as a <i>Sync</i> channel; it accepts the data item iff it can dispense it simultaneously through its sink end.
	A <i>Transformer</i> acquires data at its source end, applies a predefined transform function f on it and simultaneously writes the result to its sink end, iff the data item is in the domain of the function f . Otherwise, the channel loses the data item.
	If a $FIFO_1$ is empty, it accepts incoming data from its source end and buffers it. Being full, the channel is ready to dispense data through its sink end and become empty. Because this channel has a buffer capacity of one data item, it is either full or empty at any given time, and thus the ends of this channel cannot interact simultaneously.

constraint automata with state memory. In Section 4, we formalize the mentioned semantic model of Reo in systems of constraints along with techniques to solve them. In Section 5, we show how we abstract from the internal ports in a Reo network in a minimized representation. In Section 6, we briefly discuss some properties of our presented constraint encoding of Reo networks. Finally, in Section 7, we conclude the paper and outline our future work.





II. REO

Reo [4] is an exogenous channel-based coordination language, which can act as “glue code” to tie together software components and services. The building blocks of Reo models, *connectors*, impose constraints on the data-flow in terms of synchronization, buffering, mutual exclusion, etc. Every connector contains some *primitives*. The set of Reo primitives is open-ended, meaning that a user can define new primitives and extend the expressiveness of Reo.

The simplest Reo connector is a *channel* that has two *ends*, also called *ports*. Channel ends are either of type *source* that reads data into the channel or *sink* that writes the channel’s data out. Table I shows the most commonly used channels in Reo.

Reo nodes connect channels to each other to form Reo connectors, also called *circuits* or *networks*. Depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of both, nodes become *source*, *sink* or *mixed* nodes. A source node behaves as a synchronous *Replicator* that replicates the incoming data

Table II: Graphical Representation of Reo Nodes and Two Frequently Used Components

	A <i>Replicator</i> replicates the incoming data of its source to its sink ends simultaneously.
	A <i>Merger</i> non-deterministically chooses one of its source ends that is ready to communicate, take its incoming data item and writes it to its sink end.
	A <i>Router</i> accepts data from its source end and simultaneously writes it on one of its non-deterministically chosen sink ends that is ready to accept the data.
	A <i>Cross-product</i> reads one incoming data item from each of its incoming source ends, forms a tuple in which the data elements are set in the counter-clock-wise order with respect to its sink node, and simultaneously writes the resulting tuple on its sink end.

of its source to its sink ends simultaneously, while a sink node acts as a non-deterministic *Merger* that combines the flows of its source ends to its sink end. A mixed node is an atomic combination of a replicator and a non-deterministic merger. Each read and write action needs all of its involved source and sink ends to be able to interact synchronously; otherwise, the action cannot take place. Reo also allows hierarchical modeling and abstraction from inner structures by means of components. A component can be written in any programming language. Moreover, a connector can be converted into a component, exhibiting (part of) its inner logic as an observable behavioral interface. Table II shows the graphical representation of the Reo nodes and two frequently used components, *Router* and *Cross-product*.

A. Formal Semantics of Reo

Coordination patterns that a Reo network imposes on data-flow define the network’s *behavior*. Formal analysis of a Reo network requires formal modeling of its behavior. The behavior of a Reo network consists of various dimensions that involve:

- *State*: The states of elements forming a Reo network at any point in time define the state, also called *configuration*, of the network. The network state affects and is affected by the data-flow at each step.
- *Synchronization/Exclusion*: The term *synchronization* refers to atomic concurrent data-flow through ports and nodes. Depending on its constituents and their arrangement inside the network, in each configuration, a Reo network allows, requires, or forbids a group of ports to synchronize. We consider *mutual exclusion* as a special case of synchronization.
- *Data-dependent flow*: The value of data items that the ports of a Reo network exchange can affect the network behavior, particularly if the network contains elements, such as filter or transformer channels, that allow or forbid exchange of special data values.
- *Context dependency*: The choices some Reo elements can take change non-monotonically as the context changes [12]. A context-dependent semantics of Reo

defines context in terms of presence or absence of pending I/O requests on boundary primitive ends. As an example of such behavior consider the original description of a *LossySync* channel, which writes on its sink end the data item it reads from its source end. The channel can lose a data item only if its sink end is not write-enabled [4].

- *Timing*: The behavior of a Reo network can be subject to time constraints. For example, we can formalize a deadline for availability of some data using a $FIFO_1$ channel, which associates an expiration time with the data that it buffers. If the channel's sink end does not dispense a buffered data item before its expiration, the channel loses it [13].
- *Priority*: Presence of a prioritized element in a Reo network can influence some non-deterministic synchronization choices in the network by favoring data-flow through ports related to that prioritized element.

B. Related work and motivation

Several formal semantic models have been proposed to specify the behavior of Reo. An extensive overview and comparison of these models is presented in [7].

Connector coloring (CC) [8] is a formal semantics for Reo that describes the behavior of a connector by assigning different colors to its ports to designate presence or absence of data-flow. CC accounts for synchronization and context dependency. This model captures context dependency by propagating negative information about the absence of data-flow inside the network.

The majority of Reo semantic models are based on automata. These semantic models allow the semantics of a large network to be constructed from the given automata for its constituents using the *product* of automata. The earliest automata-based Reo semantics is Constraint Automata (CA) [10] that accommodate synchronization, states, and data-dependent flow. Transitions in CA contain the set of synchronized ports and constraints over data that the ports exchange. An extension of CA, Constraint Automata with State Memory (CASM) [11] elaborates on states by introducing state memory cells and extends data constraints to accommodate them.

Extending CA with clock assignments and timing constraints, Timed Constraint Automata (TCA) [13] express the time-aware aspect of Reo networks. A SAT-based approach for bounded model checking of TCA is presented in [14]. Another extension of CA, Constraint Automata with Priority (CAP) [15] supports the propagation of prioritized requests.

The most recent semantics of Reo [9] that is based on constraint solving, deals with synchronization, data-, and context-dependency. It uses different constraints for each of these notions, which are added conjunctively to capture their composition. This approach is the basis of the *DREAMS* [9] execution engine for Reo. Although in theory, this semantics

accounts for data-dependency, the proposed implementation ignores data. This leads to incorrect results when dealing with data-sensitive elements such as *Filter* channel.

A translation of CA and CC to a process algebraic specification language called mCRL2 [16] has been done [17]. The mCRL2 toolset compiles the specifications into Labeled Transition Systems (LTS)s, which can be verified using the mCRL2 toolset. This approach represents certain aspects of existing semantic models for Reo in an automata-based form. For instance, it models the context dependency in the behavior of Reo networks using LTSs with labels corresponding to colors in CC.

Despite the abundance of the semantics that capture different aspects of Reo, there are expressiveness gaps among them. For instance, a network with priority dependent and time sensitive behavior cannot be readily expressed by any of the mentioned formal semantics. In this paper, we present a unified symbolic constraint-based framework, where these different semantics coexist. This framework encodes the behavior of Reo networks in terms of constraints whose solutions form an existing automata-based semantic model of Reo. Unlike [17], our framework treats data symbolically, so that the state space does not blow up. Our framework extends *DREAMS* with time and priority. It also provides tool support using existing constraint solving tools to incorporate data and time constraints.

For a given configuration of a Reo network, *DREAMS* uses constraint solving to compute only a single solution for synchronous data-flow in this configuration, which leads to the next configuration. In contrast, our approach considers all solutions and configurations, which in turn enable us to map the solutions to an automata-based semantics that is well suited for model checking.

Generating the CA corresponding to a Reo network from the CA of its constituents using the CA *product* operator [10], is computationally expensive and memory inefficient. The complexity of composing m CA each having n transitions is $\mathcal{O}(n^m)$. However, we can convert this problem to the *NP*-complete problem of SAT-solving for which many efficient solvers exist.

Contribution: Instead of generating the CA corresponding to a Reo network directly by composing the CAs of its constituents, we encode each constituent CA into constraints whose conjunction forms the system of constraints that captures the full behavior of the network. The solutions of this system of constraints describe the behavior of the network in terms of variables representing different aspects of the behavior of Reo, such as synchronization, data-dependent flow, etc. From these solutions, we generate the CA corresponding to the network. We show that our solution is more efficient compared to the previous attempts in generating the formal semantics of a Reo network.

We have developed a tool to automate our approach that is integrated in the Extensible Coordination Tools (ECT)

[18]. ECT is a framework to design, develop, model check, test, and execute component-based software modeled by the Reo coordination language. The tools in this framework are integrated as Eclipse plug-ins and operate based on the operational semantics of Reo, most notably, connector coloring and constraint automata. Our tool is the only tool that supports propagation of priorities in Reo networks. Furthermore, it unifies various behavioral aspects of Reo networks in a single encoding, which enables analysis of networks whose behavior spans over several existing semantic models.

III. CONSTRAINT AUTOMATA WITH STATE MEMORY

Constraint Automata with State Memory (CASM) [11] extends CA with variables that represent local memory cells of the states of the automata. Due to its elaboration on state information, we choose CASM as the main semantic model of Reo that our framework generates.

Definition 3.1 (Constraint Automaton with State Memory):

A constraint automaton with state memory (CASM) is a tuple $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ where

- Q is a finite set of states.
- \mathcal{N} is a finite set of names.
- \rightarrow is a finite subset of $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}, \mathcal{M}, \mathcal{D}) \times Q$ is the transition relation of A , where $DC(\mathcal{N}, \mathcal{M}, \mathcal{D})$ is the set of data constraints, defined below.
- $q_0 \in Q$ is an initial state.
- \mathcal{M} is a set of memory cell names, where $\mathcal{N} \cap \mathcal{M} = \emptyset$.

Every $n \in \mathcal{N}$ represents a node in a Reo connector. The set \mathcal{N} is partitioned into three mutually disjoint sets of source nodes \mathcal{N}^{src} , mixed nodes \mathcal{N}^{mix} , and sink nodes \mathcal{N}^{snk} . Because we make the replication and merge inherent in Reo nodes explicit as *replicator* and *merger* primitives (in Table II), at most two primitive ends coincide on every node $n \in \mathcal{N}$. Thus, it follows that a source or a sink node contains only a single (source or sink) primitive end, and a mixed node contains exactly one source and one sink primitive ends.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. For every transition $q \xrightarrow{N,g} p$, we require that $g \in DC(\mathcal{N}, \mathcal{M}, \mathcal{D})$, where \mathcal{D} is the global set of numerical data values and $DC(\mathcal{N}, \mathcal{M}, \mathcal{D})$ is the language defined by the following grammar:

$$g ::= true \mid \neg g \mid g \wedge g \mid u = u \mid u < u, \\ u ::= d(n) \mid m' \mid m \mid v.$$

In this grammar, $=$ is the symmetric equality relation, $<$ is a total order relation, $n \in \mathcal{N} \subseteq \mathcal{N}$ denotes a node name, $d(n)$ represents the data item exchanged through the node n , $m \in \mathcal{M}$ correspond to a memory cell in the current state, which is the source state of the transition, m' stands for the memory cell $m \in \mathcal{M}$ in the next state, which is the target state of the transition, and $v \in \mathcal{D}$. As usual, *false* stands for

¬true, $x > y$ stands for $y < x$, and other logical operators, such as \vee and \Rightarrow (the implication symbol) can be built from the given operators.

Transitions with data constraints that can be reduced to *false* using the Boolean laws are impossible and we omit them. A data constraint g that is always *true* can be left out. We use \mathcal{M}_g to represent the set of all $m \in \mathcal{M}$ that syntactically appear as m in a data constraint g ; and \mathcal{M}'_g to refer to the set of all $m \in \mathcal{M}$ that syntactically appear as m' in g . The valuation function $\mathcal{V}_q : \mathcal{M} \rightarrow 2^{\mathcal{D}}$ designates the set of values $\mathcal{V}_q(m)$ of a memory cell $m \in \mathcal{M}$ in a state $q \in Q$, where $\mathcal{V}_{q_0}(m) = \emptyset$ for all $m \in \mathcal{M}$.

A transition $q \xrightarrow{N,g} p$ in a given constraint automaton with state memory is possible only if there exists a substitution for every syntactic element $d(n)$, m , and m' that appears in g to satisfy g . A substitution simultaneously replaces in g :

- every occurrence of $d(n)$ with the data value exchanged through the node $n \in \mathcal{N}$;
- every occurrence of m' of every $m \in \mathcal{M}$ with a value $v \in \mathcal{D}$;
- every occurrence $m \in \mathcal{M}$ with:
 - the special symbol $'\circ'$ if $\mathcal{V}_q(m) = \emptyset$
 - a value $v \in \mathcal{V}_q(m)$, otherwise.

The guard g is satisfied if proper replacement values can be found to make g *true*. Making this transition, the automaton defines the valuation function \mathcal{V}_p for the target state p , as follows: for every $m \in \mathcal{M}'_g$, $\mathcal{V}_p(m)$ is the set of all $v \in \mathcal{D}$ whose replacements for m' satisfy g . For every other $m \in \mathcal{M}$, $\mathcal{V}_p(m) = \emptyset$.

A relational operator evaluates to *true* only if the values of its operands are in its respective relation. Thus, any operator with one or more \circ as an operand always evaluates to *false*. We call a CASM, *normalized* iff a) it does not have two states with the same set of state memory variables, and b) every two transitions differ at least in their start states, their target states, or their sets of synchronizing ports. For any arbitrary CASM that is not normalized, we can normalize it by a) introducing auxiliary variables, to make the set of state memory variables unique for each state, and b) by merging the transitions that have the same start and target states and synchronize the same ports. In the sequel, we consider only *normalized* CASMs.

Following are the definitions for *product* and *hiding* operations on CASM. Both definitions are adapted from [10].

Definition 3.2 (Product-automaton): For the CASMs $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, q_{0,1}, \mathcal{M}_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, q_{0,2}, \mathcal{M}_2)$, their product is defined as:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, q_{0,1} \times q_{0,2}, \mathcal{M}_1 \cup \mathcal{M}_2)$$

where the following rules define the transition relation \rightarrow :

$$\frac{q_1 \xrightarrow{N_1, g_1} p_1, q_2 \xrightarrow{N_2, g_2} p_2, N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

$$\frac{q_1 \xrightarrow{N_1, g_1} p_1, N_1 \cap N_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N, g} \langle p_1, q_2 \rangle} \quad \frac{q_2 \xrightarrow{N_2, g_2} p_2, N_1 \cap N_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N, g} \langle q_1, p_2 \rangle}$$

We can abstract from the data-flow on certain Reo nodes using the *hiding* operator defined as follows:

Definition 3.3 (Hiding): Let $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ be a constraint automaton and $C \in \mathcal{N}$. The constraint automaton that results from hiding the node C in automaton \mathcal{A} is $\exists C[\mathcal{A}] = (Q, \mathcal{N} \setminus \{C\}, \rightarrow_C, q_0, \mathcal{M})$ and the transition relation \rightarrow_C is defined as follows:

$$\frac{p \xrightarrow{N, g} q, N' = N \setminus \{C\}, g' = \exists C[g]}{p \xrightarrow{N', g'}_C q}, \text{ where}$$

$$\exists C[g] = \bigvee_{d \in \mathcal{D}} g[d(C)/d].$$

IV. CONNECTOR COLORING

The connector coloring semantics [8] denotes the existence or absence of data-flow through the primitive ends by marking them with different colors. Let $Colors$ be the set of colors. A set of two colors, $Colors = \{-, -\}$, where $-$ denotes an occurrence and $-$ represents an absence of data-flow is adequate to express the formal semantics of many Reo networks. However, this two-color set cannot express the semantics of some Reo networks.

A traditional example of such a network is when the sink end of a *LossySync* channel connects to an empty *FIFO*₁ channel; in this case, the semantics of this network according to the two-color set includes the case where the *LossySync* loses its incoming data item, while the *FIFO*₁ channel is empty. This is an unacceptable behavior for a so-called context sensitive *LossySync* channel: it must lose its incoming data only if its sink end cannot dispense it. In the sequel, when we refer to a *LossySync* we mean its context sensitive version.

The three coloring semantics, $Colors = \{-, \triangleleft, \triangleright\}$, addresses this problem by propagating negative information regarding the absence of data-flow: it replaces $-$ with \triangleleft and \triangleright meaning that the associated primitive end, respectively, *provides* or *requires* a reason for no-flow. Considering that no-flow can occur only when at least one of the involved primitive ends *provides* a reason for it, and that an empty *FIFO*₁ cannot *provide* a reason for no-flow on its source end, the invalid behavior described above does not arise in the three coloring semantics.

Definition 4.1 (Coloring): A coloring $l : \mathcal{P} \rightarrow Colors$ is a total function from the primitive ends to a set of colors. We refer to the global set of colorings as \mathcal{L} .

Definition 4.2 (Coloring Composition): The composition of colorings l_1 and l_2 , denoted $l_1 \bullet l_2$, is defined as:

$$l_1 \bullet l_2 = \{c_1 \cup c_2 \mid c_1 \in l_1, c_2 \in l_2, p_1 \in dom(c_1), p_2 \in dom(c_2), p_1 \text{ and } p_2 \text{ are the source and sink ends of a node } n, \neg(c_1(p_1) = \triangleleft \wedge c_2(p_2) = \triangleright)\}$$

Definition 4.3 (Next function): The next function $\eta : \mathcal{L} \rightarrow 2^{\mathcal{L}}$ maps a coloring to a set of colorings, which can succeed it.

Definition 4.4 (Coloring Semantics): A coloring semantics of a Reo network is a tuple $CC = \langle \mathcal{P}, 2^{\mathcal{L}}, l_0, \eta \rangle$, where:

- \mathcal{P} is the set of primitive ends,
- $l_0 \in \mathcal{L}$ is the initial set of possible colorings,
- $2^{\mathcal{L}}$ is a set of colorings,
- η is a next function that maps a coloring to a set of colorings.

V. REO CONSTRAINT SATISFACTION PROBLEM

In Section II-A, we presented an overview of the various behavioral dimensions of a Reo network. We extend the constraint-based framework in [9] to incorporate all behavioral dimensions addressed by various semantic models for Reo. In our framework, we denote each of these elements by variables over their proper domains. We relate these variables to each other and restrict possible values they can assume using constraints whose solutions give the underlying formal semantics of the network. In the sequel, we deal only with networks whose semantics can be expressed in CASM or CC. However, we are currently extending our framework to also support timing and priority.

Let $\mathcal{N} = \mathcal{N}^{src} \cup \mathcal{N}^{mix} \cup \mathcal{N}^{snk}$ be the global set of nodes, \mathcal{M} the global set of state memory variables, and \mathcal{D} the global set of numerical data values. The set of primitive ends \mathcal{P} consists of all primitive ends p derived from \mathcal{N} by marking its elements with superscripts c and k , according to the following grammar:

$$p ::= r^c \mid s^k$$

where $r \in \mathcal{N}^{src} \cup \mathcal{N}^{mix}$ and $s \in \mathcal{N}^{snk} \cup \mathcal{N}^{mix}$. Observe that the primitive ends n^c and n^k connect on the common node n .

Let $p \in \mathcal{P}$, $n \in \mathcal{N}$ and $m \in \mathcal{M}$ be a primitive end, a node, and a state memory variable, respectively. A free variable v that occurs in the constraints encoding the behavior of a Reo network has one of the following forms:

- \tilde{n} ranges over $\{\top, \perp\}$ to show presence or absence of flow on the node n .
- \hat{n} ranges over \mathcal{D} to represent the data value passing through the node n .
- \hat{m}, \hat{m}' range over $\{\top, \perp\}$ to denote whether or not the state memory variable m is defined in, respectively, the source and the target states of the transition to which the encoded guard belongs.
- \hat{m}, \hat{m}' range over \mathcal{D} to represent the values of the state memory variable m in, respectively, the source and the target states of the transition to which the encoded guard belongs.
- \vec{p} ranges over $\{\top, \perp\}$ to state that the reason for lack of data-flow through the primitive end p originates

from, respectively, the primitive to which p belongs or the context (of this primitive).

Note that not all of the introduced variables are required for encoding the behavior of every Reo network. In presence of context dependent primitives like *LossySync* or in priority-sensitive networks, constraints include variables of the form \vec{p} . For the stateful elements such as *FIFO*₁, variables like $\hat{m}, \hat{m}', \hat{m}$, and \hat{m}' appear in the constraints.

Observe that the interpretation of some of the mentioned variables depends on the values of other variables. Referring to the variable \vec{p} makes sense only if $\hat{n} = \perp$, where $p = n^c$ or $p = n^k$ (i.e., the primitive end p belongs to the node n); and \hat{n}, \hat{m} and \hat{m}' make sense only if $\hat{n} = \top, \hat{m} = \top$ and $\hat{m}' = \top$, respectively.

The grammar for a constraint Ψ encoding the behavior of a Reo network is as follows:

$$\begin{aligned} t &::= \hat{n} \mid \hat{m} \mid \hat{m}' \mid d \mid t \otimes d && \text{(terms)} \\ a &::= \hat{n} \mid \vec{p} \mid \hat{m} \mid \hat{m}' \mid t = t \mid t < t && \text{(atoms)} \\ \Psi &::= a \mid \neg\Psi \mid \Psi \wedge \Psi && \text{(formulae)} \end{aligned}$$

where $d \in \mathcal{D}$ is a constant, $\otimes \in \{+, -, *, /, \%, \wedge\}$, and p is either of the form n^c or n^k .

Definition 5.1 (Reo Constraint Satisfaction Problem):

A Reo Constraint Satisfaction Problem (RCSP) is a tuple $\langle \mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$, where:

- \mathcal{P} is a finite set of primitive ends.
- \mathcal{M} is a finite set of state memory variables.
- $M_0 \subseteq \mathcal{M}$ is a set of state memory variables that define the initial configuration of a Reo network.
- \mathcal{V} is a set of variables v defined by the grammar

$$v ::= \hat{n} \mid \vec{p} \mid \hat{m} \mid \hat{m}' \mid \hat{n} \mid \hat{m} \mid \hat{m}'$$

for $n \in \mathcal{N}, p \in \mathcal{P}$, and $m \in \mathcal{M}$. The values that the variables of the forms \hat{n}, \hat{m} , and \hat{m}' can assume are subsets of \mathcal{D} , and the other variables are Boolean, with values in $\{\top, \perp\}$.

- $C = \{C_1, C_2, \dots, C_m\}$ is a finite set of constraints, where each C_i is a constraint given by the grammar Ψ involving a subset of variables $V_i \subseteq \mathcal{V}$.

Example 5.1: The RCSP of a *Sync* channel with the source end a and the sink end b is $\langle \{a, b\}, \emptyset, \emptyset, \{\hat{a}, \hat{b}, \hat{a}, \hat{b}\}, \hat{a} \Leftrightarrow \hat{b} \wedge \hat{a} \Rightarrow (\hat{a} = \hat{b}) \rangle$. The solutions for this constraint problem give the behavior of the *Sync* channel as the channel allows data-flow on its source end iff its sink end can dispense it simultaneously (which agrees with the semantics of this channel as defined in other formal models of Reo). In case of data-flow, the values of the data items passing through the ends of this channel are equal.

We obtain the constraints corresponding to a Reo network by composing the RCSPs of its constituents as defined below.

Definition 5.2 (Composition): The composition of two RCSPs $\rho_1 = \langle \mathcal{P}_1, \mathcal{M}_1, M_{0,1}, \mathcal{V}_1, C_1 \rangle$ and $\rho_2 = \langle \mathcal{P}_2, \mathcal{M}_2, M_{0,2}, \mathcal{V}_2, C_2 \rangle$ is defined as follows:

$$\rho_1 \odot \rho_2 = \langle \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{M}_1 \cup \mathcal{M}_2, M_{0,1} \cup M_{0,2}, \mathcal{V}_1 \cup \mathcal{V}_2, C_1 \wedge C_2 \rangle$$

However, connecting two Reo networks must not introduce incorrect data-flow possibilities. This is done by enforcing a restriction on the possible solutions through the following axiom:

Axiom 1 (Mixed node axiom): When two Reo networks connect on the common node x , where x^c is a source end in one network and x^k is a sink end in the other, the following constraint must hold:

$$\neg \vec{x} \Leftrightarrow (\vec{x}^c \vee \vec{x}^k)$$

The *mixed node axiom*, which applies to all mixed nodes in a network, states that a node x cannot produce the reason for no-flow all by itself.

A. Encoding Reo Elements in RCSPs

Table III summarizes the constraint encodings associated with commonly used Reo elements. If a Reo network does not contain any context dependent channel, the variables encoding the context dependency can be ignored in its RCSP. Table IV shows the encoding of Reo elements from Table III where the context variables are removed. Note that in these tables, a and b denote the source and the sink ends of a primitive, respectively, and that *dom* refers to the domain of the given function or predicate. In the case of elements with more than one source or sink ends, we use indices.

The intuition behind these constraints is that their solutions reflect the semantic model of each element as given by CASM and CC.

Example 5.2: Figure 1 shows a Reo network that consists of a *transformer* channel with the function $3 * \hat{a}$, whose domain is the set of numbers *Number* and a *filter* channel with the condition $\hat{b} \% 2 = 0$ and domain *Number*.

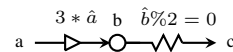


Figure 1: A Data-Aware Reo Network

Since none of the Reo primitives in Figure 1 is context dependent, we use the constraints corresponding to the primitives in this network as defined in Table IV.

Equation 1 states that flow occurs on the source end of the *transformer* channel iff it occurs on its sink end. In addition, flow can exist only if the data item that enters the source end of the channel is a number. In this case, the data item written on the sink end is three times the value of the source data item.

Equation 2 expresses that flow on the source end of the *filter* channel leads to flow on its sink end, iff the data item belongs to the channel's accepting pattern (which is

Table III: Context Dependent Encoding of Reo Primitives (Extending [9])

Channel	Constraints
	$\psi_{Sync}(a, b) : \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} = \hat{b}) \wedge \neg(\overline{a^c} \wedge \overline{b^k})$
	$\psi_{SyncDrain}(a_1, a_2) : \tilde{a}_1 \Leftrightarrow \tilde{a}_2 \wedge \neg(\overline{a_1^c} \wedge \overline{a_2^c})$
	$\psi_{AsyncDrain}(a_1, a_2) : \tilde{a}_1 \Rightarrow (\neg\tilde{a}_2 \wedge \overline{a_2^c}) \wedge \tilde{a}_2 \Rightarrow (\neg\tilde{a}_1 \wedge \overline{a_1^c})$
	$\psi_{LossySync}(a, b) : \tilde{b} \Rightarrow \tilde{a} \wedge \tilde{b} \Rightarrow (\hat{a} = \hat{b}) \wedge \neg\overline{a^c} \wedge \neg\overline{b^k} \Rightarrow \overline{b^k}$
	$\psi_{Merger}(a_{0..i}, b) : \tilde{a}_i \Leftrightarrow \tilde{b} \wedge \tilde{a}_i \Rightarrow (\hat{a}_i = \hat{b}) \wedge \neg\tilde{b} \Rightarrow ((\neg\overline{b^k} \wedge \overline{a_i^c}) \vee (\overline{b^k} \wedge \neg\overline{a_i^c} \wedge \bigwedge_{j,j \neq i} \overline{a_j^k}))$
	$\psi_{Replicator}(a, b_{0..i}) : \tilde{a} \Leftrightarrow \bigwedge_i \tilde{b}_i \wedge (\tilde{a} \Rightarrow \bigwedge_i (\hat{b}_i = \hat{a})) \wedge \neg\tilde{a} \Rightarrow ((\neg\overline{a^c} \wedge \overline{b_i^k}) \vee (\overline{b_i^k} \wedge \bigwedge_{j,j \neq i} \overline{b_j^k} \wedge \overline{a^c}))$
	$\psi_{Router}(a, b_{0..i}) : \tilde{a} \Leftrightarrow (\bigvee_i \tilde{b}_i) \wedge \bigwedge_{j,j \neq i} \neg(\tilde{b}_i \wedge \tilde{b}_j) \wedge \tilde{b}_i \Rightarrow (\hat{b}_i = \hat{a}) \wedge \tilde{a} \Rightarrow (\neg\overline{a^c} \vee \neg(\bigvee_i \overline{b_i^k}))$
	$\psi_{FIFO1}(a, b, m) : \tilde{a} \Rightarrow (\neg\hat{m} \wedge \hat{m}' \wedge (\hat{m}' = \hat{a})) \wedge \tilde{b} \Rightarrow (\hat{m} \wedge \neg\hat{m}' \wedge (\hat{m} = \hat{b})) \wedge (\neg\tilde{a} \wedge \neg\tilde{b}) \Rightarrow (\hat{m} \Leftrightarrow \hat{m}' \wedge \hat{m} \Rightarrow (\hat{m} = \hat{m}')) \wedge \neg\hat{m} \Rightarrow \overline{b^k} \wedge \hat{m} \Rightarrow \overline{a^c}$
	$\psi_{Filter}(a, b, P) : \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{a} \in \text{dom}(P) \wedge P(\hat{a})) \wedge \tilde{b} \Rightarrow (\hat{a} = \hat{b}) \wedge (\neg\tilde{a} \Rightarrow (\neg\overline{a^c} \Leftrightarrow \overline{b^k})) \wedge (\tilde{a} \wedge \neg\tilde{b} \Rightarrow \overline{b^k})$
	$\psi_{Transformer}(a, b, f) : \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{a} \in \text{dom}(f)) \wedge \tilde{b} \Rightarrow (\hat{b} = f(\hat{a})) \wedge \neg(\overline{a^c} \wedge \overline{b^k})$

Table IV: Encoding Reo Primitives (Extending [9])

Channel	Constraints
	$\psi_{Sync}(a, b) : \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} = \hat{b})$
	$\psi_{SyncDrain}(a_1, a_2) : \tilde{a}_1 \Leftrightarrow \tilde{a}_2$
	$\psi_{AsyncDrain}(a_1, a_2) : \neg(\tilde{a}_1 \wedge \tilde{a}_2)$
	$\psi_{LossySync} : \tilde{b} \Rightarrow \tilde{a} \wedge \tilde{b} \Rightarrow (\hat{a} = \hat{b})$
	$\psi_{Merger}(a_{0..i}, b) : \tilde{b} \Leftrightarrow (\bigvee_i \tilde{a}_i) \wedge \bigwedge_{j,j \neq i} \neg(\tilde{a}_i \wedge \tilde{a}_j) \wedge \tilde{a}_i \Rightarrow (\hat{a}_i = \hat{b})$
	$\psi_{Replicator}(a, b_{0..i}) : \tilde{a} \Leftrightarrow (\bigwedge_i \tilde{b}_i) \wedge \tilde{a} \Rightarrow (\bigwedge_i (\hat{b}_i = \hat{a}))$
	$\psi_{Router}(a, b_{0..i}) : \tilde{a} \Leftrightarrow (\bigvee_i \tilde{b}_i) \wedge \bigwedge_{j,j \neq i} \neg(\tilde{b}_i \wedge \tilde{b}_j) \wedge \tilde{b}_i \Rightarrow (\hat{b}_i = \hat{a})$
	$\psi_{FIFO1}(a, b, m) : \tilde{a} \Rightarrow (\neg\hat{m} \wedge \hat{m}' \wedge (\hat{m}' = \tilde{a})) \wedge \tilde{b} \Rightarrow (\hat{m} \wedge \neg\hat{m}' \wedge (\hat{m} = \tilde{b})) \wedge (\neg\tilde{a} \wedge \neg\tilde{b}) \Rightarrow (\hat{m} \Leftrightarrow \hat{m}' \wedge \hat{m} \Rightarrow (\hat{m} = \hat{m}'))$
	$\psi_{Filter}(a, b, P) : \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{b} \in \text{dom}(P) \wedge P(\hat{a})) \wedge (\hat{a} = \hat{b})$
	$\psi_{Transformer}(a, b, f) : \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{b} \in \text{dom}(f)) \wedge \tilde{b} \Rightarrow (\hat{b} = f(\hat{a}))$

$$\psi_{Transformer}(a, b, 3*\hat{a}) = \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} \in \text{Number} \wedge \hat{b} = 3*\hat{a}) \quad (1)$$

$$\psi_{Filter}(b, c, \hat{b}\%2 = 0) = \tilde{c} \Rightarrow (\tilde{b} \wedge \hat{b} \in \text{Number} \wedge (\hat{b}\%2 = 0)) \quad (2)$$

$\hat{b}\%2 = 0$). In this case, the value of data items passing through the ends are equal. No flow through the sink end c is either due to no flow on b or that the incoming data item does not satisfy the accepting pattern. As mentioned, the conjunction of these constraints (subject to Axiom 1, which trivially holds in this case) encodes the behavior of the given Reo network.

B. Solving RCSPs

In this section, we formalize the solutions of RCSPs and show how to obtain them.

Definition 5.3 (Solution): A solution \mathcal{S} for a constraint C is a function $\mathcal{S} : \mathcal{V} \rightarrow 2^{\mathcal{D}} \cup \{\top, \perp\}$ such that for all distinct $v_i \in \mathcal{V}, 1 \leq i \leq n = |\mathcal{V}|$, we have $z_i \in \mathcal{S}(v_i)$ implies $C[v_1, v_2, \dots, v_n \setminus z_1, z_2, \dots, z_n]$ is true.

Since Reo Constraint Satisfaction Problems (RCSPs) have predicates with free variables of types Boolean ($\{\top, \perp\}$) and data (\mathcal{D}), a SAT-solver or a numeric constraint solver cannot solve them alone. Satisfiability Modulo Theories (SMT) [19] solvers find solutions for propositional satisfiability problems where propositions are either Boolean or constraints in a specific theory. However, SMT-solvers are not applicable in our case either, because unlike SAT-solvers they find only an instance of a solution as opposed to the complete set of answers. Another drawback of most SAT- and SMT-solvers is that they work only on quantifier-free formulae, while we use existential quantifiers to implement the *hiding* operator of Constraint Automata (see Section VI).

To generate the CASM corresponding to a given Reo network, we need all solutions and thus resort to a hybrid approach that uses both SAT-solvers and Computer Algebra Systems (CASs), namely, REDUCE [20], which is a system for general algebraic computations. First, we form a pure Boolean constraint system by substituting data dependent constraints with new Boolean variables. We find all solutions for the new constraints using a SAT-solver. Then, by substituting each such solution into the original constraints, we obtain a data dependent constraint satisfaction problem that a CAS can solve symbolically. From these solutions, we extract a CASM corresponding to the Reo network encoded by the original set of constraints. Our approach avoids state explosion by treating data constraints symbolically. In the following, we elaborate on our approach.

In an RCSP $\langle \mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$, let \mathcal{V}_B and \mathcal{V}_D be the sets of free Boolean and free data variables of C , respectively, where $\mathcal{V} = \mathcal{V}_B \cup \mathcal{V}_D$, and let A_D be the set of atomic predicates of C containing data variables. The following is our procedure for solving C .

- 1) We obtain C_B from C by replacing every occurrence of $x \in A_D$ with a unique new Boolean variable $y \notin \mathcal{V}$. For example, for $C = (\tilde{c} \Rightarrow \tilde{b}) \wedge (\tilde{c} \Rightarrow (\hat{b} \in \text{Number} \Rightarrow \hat{b}\%2 = 0))$ in Example 5.2, we obtain C_B as $(\tilde{c} \Rightarrow \tilde{b}) \wedge (\tilde{c} \Rightarrow (y_1 \Rightarrow y_2))$ where y_1 and y_2 replace $\hat{b} \in \text{Number}$ and $\hat{b}\%2 = 0$, respectively.
- 2) An off-the-shelf SAT-solver can find the set of solutions S_B for C_B . We define the finite set of constraints $C[S_B] = \{C[v_1, v_2, \dots, v_n \setminus z_1, z_2, \dots, z_n] \mid \text{for all distinct } v_i \in \mathcal{V}_B, 1 \leq i \leq n = |\mathcal{V}_B|, z_i \in S(v_i), S \in S_B\}$.
- 3) Every $C_D \in C[S_B]$ is a numerical constraint satisfaction problem, which we (symbolically) solve using

a Computer Algebra System. Every solution to each C_D along with the SAT solution $S \in \mathcal{S}_B$ that produced $C_D \in C[S_B]$ in the previous step, constitute a solution to the RCSP.

Using the presented technique, we obtain the solutions for the RCSP corresponding to Examples 5.2 as follows:

- 1) $\langle \{\bar{a} = \perp, \bar{b} = \perp, \bar{c} = \perp\}, \top \rangle$,
- 2) $\langle \{\bar{a} = \top, \bar{b} = \perp, \bar{c} = \perp\}, \hat{a} \notin \text{Number} \rangle$,
- 3) $\langle \{\bar{a} = \top, \bar{b} = \top, \bar{c} = \perp\}, \hat{a} \in \text{Number} \wedge \hat{b} = 3 * \hat{a} \wedge \hat{b} \% 2 \neq 0 \rangle$,
- 4) $\langle \{\bar{a} = \top, \bar{b} = \top, \bar{c} = \top\}, \hat{a} \in \text{Number} \wedge \hat{b} = 3 * \hat{a} \wedge \hat{b} \% 2 = 0 \wedge \hat{b} = \hat{c} \rangle$.

Example 5.3: Figure 2 depicts a Reo network that consists of a *LossySync* channel and a *FIFO₁* channel connecting on the node b .

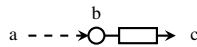


Figure 2: A Context Sensitive Reo Network

Since the Reo network in Figure 2 contains a *LossySync* that is a context dependent channel, we use the context-aware RCSP encoding from Table III:

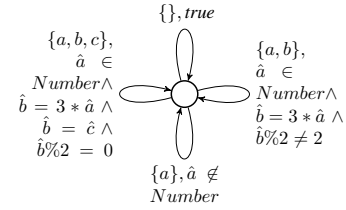
$$\psi_{\text{LossySync}}(a, b) = \bar{b} \Rightarrow (\bar{a} \wedge (\hat{a} = \hat{b})) \wedge \neg \bar{a} \wedge \neg \bar{a} \Rightarrow \bar{b}^k. \quad (3)$$

$$\psi_{\text{FIFO}_1}(b, c, m) = \bar{b} \Rightarrow (\neg \hat{m} \wedge \hat{m}' \wedge (\hat{m}' = \hat{b})) \wedge \bar{c} \Rightarrow (\hat{m} \wedge \neg \hat{m}' \wedge (\hat{m} = \hat{c})) \wedge (\neg \bar{b} \wedge \neg \bar{c}) \Rightarrow ((\hat{m} \Leftrightarrow \hat{m}') \wedge \hat{m} \Rightarrow (\hat{m} = \hat{m}')) \wedge \neg \hat{m} \Rightarrow \bar{c}^c \wedge \hat{m} \Rightarrow \bar{b}^c. \quad (4)$$

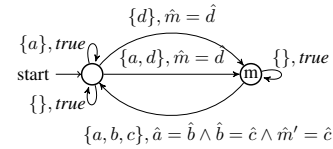
Equation 3 states that flow on the sink end of the *LossySync* is due to flow on its source end. If there is flow on the sink end of the *LossySync*, the data items exchanged at the source and the sink ends are the same. However, it is possible that the source end has flow, but the sink end does not. In this case, the reason for no flow comes from the environment with which the sink end communicates. The third possible behavior of the channel is that there is no flow on the source end due to the environment, in which case the channel provides a reason for no flow on its sink end.

Equation 4 expresses the behavior of the *FIFO₁* channel as follows: The flow on the source end of the channel states that the value of the variable representing the state memory (of the current state) is undefined. The flow on the source end defines the state memory variable for the next state to contain the value of the incoming data item. On the other hand, flow on the sink end means that the value of the state memory variable is defined. The data item leaving the sink end is equivalent to the buffer's data item. In addition, the value of the state memory variable becomes undefined in the next state. If there is no flow on the ends, the variables related to the states stay the same. Being empty, the *FIFO₁* channel provides a reason for no flow on its sink end, while being full does so on the source end of the channel.

The solutions for the RCSP of Example 5.3, (where for brevity, we omit the values of the variables representing the context, such as \bar{b}^c) are as follows:



(a)



(b)

Figure 3: CASMs Generated for Examples 5.2 (a) and 5.3 (b)

- 1) $\langle \{\bar{a} = \perp, \bar{b} = \perp, \bar{c} = \perp, \hat{m} = \perp, \hat{m}' = \perp\}, \top \rangle$,
- 2) $\langle \{\bar{a} = \top, \bar{b} = \top, \bar{c} = \perp, \hat{m} = \perp, \hat{m}' = \top\}, \hat{a} = \hat{b} \wedge \hat{m}' = \hat{b} \rangle$,
- 3) $\langle \{\bar{a} = \top, \bar{b} = \perp, \bar{c} = \perp, \hat{m} = \top, \hat{m}' = \top\}, \hat{m} = \hat{m}' \rangle$,
- 4) $\langle \{\bar{a} = \perp, \bar{b} = \perp, \bar{c} = \perp, \hat{m} = \top, \hat{m}' = \top\}, \hat{m} = \hat{m}' \rangle$,
- 5) $\langle \{\bar{a} = \top, \bar{b} = \perp, \bar{c} = \perp, \hat{m} = \top, \hat{m}' = \perp\}, \hat{m} = \hat{c} \rangle$,
- 6) $\langle \{\bar{a} = \perp, \bar{b} = \perp, \bar{c} = \top, \hat{m} = \top, \hat{m}' = \perp\}, \hat{m} = \hat{c} \rangle$.

C. CASM Construction

In order to construct the CASM from the set of solutions S for an RCSP $\langle \mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$, we first define

- $\mathcal{N} = \{n \mid n^c \in \mathcal{P} \vee n^k \in \mathcal{P}\}$

and then map each solution $\langle s, s_d \rangle \in S$ into a transition $t : q \xrightarrow{N, g} p$ as follows:

- $q = \langle \{m \mid m \in \mathcal{M}, s(\hat{m}) = \top \} \rangle$,
- $p = \langle \{m \mid m \in \mathcal{M}, s(\hat{m}') = \top \} \rangle$,
- $N = \{n \mid n \in \mathcal{N}, s(\hat{n}) = \top \}$,
- The data constraint g is (a syntactic variant of) s_d .

We obtain the CASM $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ from the set \rightarrow of all transitions generated above, where:

- $Q = \{q \mid q \xrightarrow{N, g} p \vee p \xrightarrow{N, g} q\}$,
- $q_0 = \langle \{m \mid m \in M_0, s(\hat{m}) = \top \} \rangle$,
- \mathcal{M} is the same \mathcal{M} as in the RCSP.

Applying the above procedure to the solutions of RCSPs constraints generates their corresponding CASMs. For instance, the first solution for the constraints in Example 5.2 generates the transition $q \xrightarrow{\emptyset, true} q$, where q is the only state of the CASM, which has no state memory variable. This is so because the set of variables of the form \hat{m} is empty. Also, the transition has no synchronizing port, because the value of every one of variables \bar{a}, \bar{b} and \bar{c} is \perp . Figures 3a and 3b show the CASMs derived from the RCSPs in Examples 5.2 and 5.3.

Our approach deals with data in a symbolic fashion, where we partition the global set of data values to equivalence classes toward which a Reo network behaves differently.

This is in contrast with the traditional way of dealing with data in the formal semantics of Reo (and other models), where they consider a different state for each possible value that can be stored in buffers and a distinct transition for each data value passing through the ports. Our symbolic approach allows working with an infinite data domain. In addition, rather than implementing the highly time- and memory-demanding custom-made algorithms to generate Reo formal semantics, we use the efficient SAT-solvers and computer algebra systems to solve constraints whose solutions are equivalent to these models. An experimental study on the efficiency of using SAT-solvers to generate Reo formal semantics is reported in [9].

VI. HIDING

We use *hiding* to abstract from internal transitions. The author in [9] proposes applying the existential quantifier to the constraints encoding of the behavior of a network to abstract from internal ports and their corresponding data variables. Similarly, we use existential quantifiers such as $\exists \tilde{e}, \hat{e}, \vec{e} : C$, where C is the RCSP of a Reo network and e is an internal node to hide.

Although several algorithms exist for the problem of quantifier elimination in Boolean algebra and first order logic [21], [22] and [23], we are not aware of any working tool that does quantifier elimination on Boolean algebraic formulae. Therefore, our tool implements the *hiding* operator as defined for CASM.

Hiding the internal nodes on some transitions can make the set of their synchronized nodes empty. Here, we refer to such a transition as an *empty* transition, if the free variables of its guard are merely state memory variables. Under some circumstances, we can merge the source and the target states of empty transitions. Let q and p be two states in a CASM such that $q \xrightarrow{\emptyset, g} p$. The following are the conditions under which the state p can merge into the state q :

- 1) The states q and p have the same number of state memory variables.
- 2) The guard g consists of the conjunction of the predicates of the form of $x = y'$, for $x, y \in \mathcal{M}$. This way, g defines a correspondence relation between the state memory variables of the state q and those of the state p .
- 3) For each transition $q \xrightarrow{N, g'} r$ where $r \notin \{p, q\}$, there is a transition $p \xrightarrow{N, g''} r$ such that $g' \Leftrightarrow g''$, where g'' is obtained from g by replacing all occurrences of the next state memory variable y' with the next state memory variable x' , if g contains $x = y'$ for state memory variables $x, y \in \mathcal{M}$.
- 4) For each transition $r \xrightarrow{N, g'} p$ where $r \notin \{p, q\}$, there is a transition $r \xrightarrow{N, g''} q$ such that $g' \Leftrightarrow g''$, where g'' is derived from g by substituting all occurrences of the

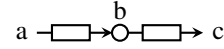
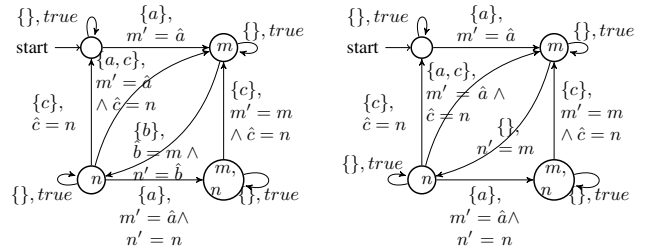
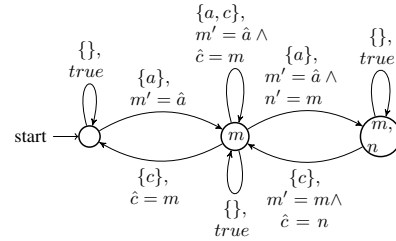


Figure 4: Two $FIFO_1$ s Forming $FIFO_2$



(a) CASM of Example 6.1

(b) Hiding Internal Ports



(c) Merging the States

Figure 5: Hiding the Empty Transition and Merging Its Source and Target States for the CASM of $FIFO_2$ in Figure 4

state memory variable x in g with the state memory variable x , if g contains $x = y'$ for state memory variables $x, y \in \mathcal{M}$.

Provided that the above conditions hold, the state p merges into the state q as follows:

- 1) We eliminate the transition $q \xrightarrow{\emptyset, g} p$.
- 2) We remove the state p after substituting y, y' , and p with x, x' , and q in all transitions. Observe that such substitutions convert the non-eliminated transitions between the states q and p into loops over the state q .

Example 6.1: Figure 4 shows a $FIFO_2$ derived from composing two $FIFO_1$ s. The CASM corresponding to the $FIFO_2$ is in Figure 5a. Figure 5b depicts the CASM resulting from hiding the mixed node b . Figure 5c presents the result of eliminating the empty transitions.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a constraint-based framework that encodes the semantics of Reo networks as constraint satisfaction problems whose predicates are either Boolean propositions or numerical constraints. We presented a hybrid approach to find the solutions for these problems. An advantage of our approach is that it treats data constraints symbolically to mitigate the state explosion problem. From this solution, we construct the semantic model corresponding

to a Reo network in the form of constraint automata with state memory. Our framework supports *product* and *hiding* operations on constraint automata. We have implemented and integrated our approach as a tool in the ECT. As part of our ongoing work, we are using this framework to encode other aspects of the semantics of Reo, namely, priorities and timed behavior. In this way, our work will be the most expressive framework that exists to analyze Reo networks. Furthermore, we will prove soundness and completeness of the RCSP encoding of Reo networks along with its compositionality.

REFERENCES

- [1] M. Bell, "Introduction to Service-Oriented Modeling," *Service-Oriented Modeling: Service Analysis, Design, and Architecture*, p. 3, 2008.
- [2] R. Pugliese and F. Tiezzi, "A Calculus for Orchestration of Web Services," *J. Applied Logic*, vol. 10, no. 1, pp. 2–31, 2012.
- [3] D. Kitchin, W. R. Cook, and J. Misra, "A Language for Task Orchestration and Its Semantic Properties," in *CONCUR*, vol. 4137 of *Lecture Notes in Computer Science*, pp. 477–491, Springer, 2006.
- [4] F. Arbab, "Reo: a Channel-Based Coordination Model for Component Composition," *Mathematical Structures in Computer Science*, vol. 14, pp. 329–366, 2004.
- [5] F. Arbab, N. Kokash, and S. Meng, "Towards Using Reo for Compliance-Aware Business Process Modeling," in *ISoLA*, pp. 108–123, 2008.
- [6] B. Changizi, N. Kokash, and F. Arbab, "A Unified Toolset for Business Process Model Formalization," in *7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010)*, pp. 147–156, ENTCS, 2010.
- [7] S.-S. T. Jongmans and F. Arbab, "Overview of Thirty Semantic Formalisms for Reo," *Scientific Annals of Computer Science*, vol. 22, pp. 201–251, 2012.
- [8] D. Clarke, D. Costa, and F. Arbab, "Connector Colouring I: Synchronisation and Context Dependency," *Science of Computer Programming*, vol. 66, no. 3, pp. 205–225, 2007.
- [9] J. Proença, *Synchronous Coordination of Distributed Components*. PhD thesis, Institute for Programming research and Algorithms, 2011.
- [10] C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten, "Modeling Component Connectors in Reo by Constraint Automata," *Science of Computer Programming*, vol. 61, no. 2, pp. 75–113, 2006.
- [11] B. Pourvatan, M. Sirjani, H. Hojjat, and F. Arbab, "Symbolic Execution of Reo Circuits using Constraint Automata," *Sci. Comput. Program.*, vol. 77, no. 7-8, pp. 848–869, 2012.
- [12] M. M. Bonsangue, D. Clarke, and A. Silva, "Automata for Context-Dependent Connectors," in *COORDINATION* (J. Field and V. T. Vasconcelos, eds.), vol. 5521 of *Lecture Notes in Computer Science*, pp. 184–203, Springer, 2009.
- [13] F. Arbab, C. Baier, F. D. Boer, and J. Rutten, "Models and Temporal Logics for Timed Component Connectors," in *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pp. 198–207, IEEE Computer Society, 2004.
- [14] S. Kemper, "SAT-based Verification for Timed Component Connectors," *Electr. Notes Theor. Comput. Sci.*, vol. 255, pp. 103–118, 2009.
- [15] F. Arbab and C. Baier, "Priority in Reo and Constraint Automata," tech. rep., Centrum voor Wiskunde en Informatica. In preparation.
- [16] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. V. Weerdenburg, "The Formal Specification Language mCRL2," in *Methods for Modelling Software Systems (MMOSS 2006)*, vol. 06351 of *Dagstuhl Seminar Proceedings*, IBFI, 2006.
- [17] N. Kokash, C. Krause, and E. de Vink, "Reo + mCRL2: A Framework for Model-checking Dataflow in Service Compositions," *Formal Aspects of Computing*, 2011.
- [18] F. Arbab, C. Koehler, Z. Maraikar, Y.-J. Moon, and J. Proença, "Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools," in *5th International Workshop on Formal Aspects of Component Software (FACS 2008)*, vol. 8, ENTCS, 2008.
- [19] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," *Handbook of Satisfiability*, vol. 4, 2009.
- [20] G. Rayna, *REDUCE: Software for Algebraic Computation*. New York, NY, USA: Springer-Verlag New York, Inc., 1987.
- [21] L. Bordeaux and L. Zhang, "A Solver for Quantified Boolean and Linear Constraints," in *Proceedings of the 2007 ACM symposium on Applied computing, SAC '07*, (New York, NY, USA), pp. 321–325, ACM, 2007.
- [22] A. Ayari and D. Basin, "QUBOS: Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers," in *Formal Methods in Computer-Aided Design*, pp. 187–201, Springer, 2002.
- [23] J. H. Davenport, "Computer Algebra Applied to Itself," *J. Symb. Comput.*, vol. 6, pp. 127–132, August 1988.