

Channel-based Coordination via Constraint Satisfaction

Dave Clarke^a, José Proença^{b,1}, Alexander Lazovik^{c,2}, Farhad Arbab^b

^a*Dept. Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium*

^b*CWI, Science Park 123, 1098 XG Amsterdam, The Netherlands*

^c*University of Groningen Postbus 800, 9700 AV Groningen, The Netherlands*

Abstract

Coordination in \mathcal{Reo} emerges from the composition of the behavioural constraints of primitives, such as channels, in a component connector. Understanding and implementing \mathcal{Reo} , however, has been challenging due to interaction of the *channel metaphor*, which is an inherently local notion, and the non-local nature of the constraints imposed by composition. In this paper, the channel metaphor takes a back seat. We focus on the behavioural constraints imposed by the composition of primitives and phrase the semantics of \mathcal{Reo} as a constraint satisfaction problem. Not only does this provide a clear description of the behaviour of \mathcal{Reo} connectors in terms of *synchronisation* and *data flow constraints*, it also paves the way for new implementation techniques based on constraint satisfaction. We also demonstrate that this approach is more efficient than existing techniques based on connector colouring.³

Key words: Coordination, Constraint Satisfaction, Constraint Automata, \mathcal{Reo} , Connector Colouring

1. Introduction

Coordination models and languages [43] are fundamental tools for reducing the intrinsic complexity of systems of concurrent, distributed, mobile and heterogeneous components, by providing glue code that acts as the linguistic counterpart of middleware. Coordination models and languages offer a layer between components to intercept, modify, redirect, and synchronise communications between components, and provide facilities to monitor and manage their resource usage, typically outside of the components themselves.

Wegner describes coordination as constrained interaction [51]. We take this idea literally and represent coordination using constraints to develop an efficient

Email addresses: dave.clarke@cs.kuleuven.be (Dave Clarke), proenca@cwi.nl (José Proença), a.lazovik@rug.nl (Alexander Lazovik), farhad@cwi.nl (Farhad Arbab)

¹Supported by FCT grant 22485 - 2005, Portugal.

²Work carried out during an ERCIM “Alain Bensoussan” Fellowship.

³This article is a revised and expanded version of Ref. [22].

implementation of the *Reo* coordination model [4]. *Reo* takes the idea of using channels to connect components and pushes this further by allowing channels to also be composed with each other, resulting in an expressive notion of component connector. Each channel imposes behavioural constraints on the entities it connects; these constraints are propagated through a connector, enabling quite complex behaviours to be expressed with the aid of a graphical notation. *Reo* has been applied in numerous areas, for instance, component-based software engineering [13], protocol modelling [52], web service composition [38], and business process modelling [11].

We will adopt the view that a *Reo* connector specifies a (series of) constraint satisfaction problems, and that valid interaction between a connector and its environment (in each state) corresponds to the solutions of such constraints. This idea diverges from the existing descriptions of *Reo*, which are based on data flow through channels, but we claim that this viewpoint not only makes it easier to understand *Reo* connectors, but also opens the door to more efficient implementation techniques—a claim supported by experimental results in Section 6 and to alternative ways of thinking about ‘channel-based’ coordination. Thus two concerns motivate our work. We use constraints to describe the semantics of *Reo* connectors in order to develop more efficient implementation techniques for *Reo*, utilising existing SAT solving and constraint satisfaction techniques. Thus, we use constraints to describe the semantics of *Reo* connectors in order to develop more efficient implementation techniques for *Reo*, utilising existing SAT solving and constraint satisfaction techniques.

The paper makes the following technical contributions beyond the state of the art:

- a constraint-based semantic description of *Reo* connectors; and
- SAT- and CSP-based implementation techniques for *Reo* connectors, which are more efficient than existing techniques.

Organisation. This paper is organised as follows. We elaborate on *Reo* in Section 2, where we also present examples of *Reo* connectors. Section 3 describes our encoding of *Reo*-style coordination as a constraint satisfaction problem. Section 4 describes an extension of this encoding to incorporate state, so that connector semantics can be completely internalised as constraints. Correctness and compositionality properties are also presented. Section 5 describes how to encode context dependency, as formulated in the connector colouring model [20], into constraints. Section 6 presents some benchmarking results comparing an existing engine for *Reo* based on connector colouring with two prototypes engines based on constraint solving techniques, with and without context dependency. Section 7 describes how to guide the underlying constraint solver to achieve fairness and priority. Section 8 presents some implementation issues, in particular, it gives a description of the alternative interaction model constraint satisfaction enables. Section 9 discusses and compares existing *Reo* models and implementations with our constraint-based approach. Finally, Sections 10 and 11 present related work and our conclusions.

2. Reo Coordination Model

Reo [4, 5] is a channel-based coordination model, wherein coordinating *connectors* are constructed compositionally out of more primitive connectors, which we call *primitives*. Primitives, including channels, offer a variety of behavioural policies regarding synchronisation, buffering, and lossiness. Primitives also include n -ary mergers and n -ary replicators (although binary mergers and replicators are sufficient) [14, 20]. Being able to compose connectors out of smaller primitives is one of the strengths of Reo. It allows, for example, multi-party synchronisation to be expressed simply by plugging simple channels together. In addition, Reo’s graphical notation helps bring some intuition about the behaviour of a connector, particularly in conjunction with animation tools.⁴

Communication with a primitive occurs through its ports, called *ends*: primitives consume data through their *source ends*, and produce data through their *sink ends*. Source and sink ends correspond to the notion of source and sink in directed graphs, although the names *input* and *output* are sometimes used instead. Primitives are not only a means for communication, but they also impose relational constraints, such as synchronisation or mutual exclusion, on the timing of data flow on their ends. The behaviour of such primitives is limited only by the model underlying a given Reo implementation.

The behaviour of each primitive depends upon its current state. The semantics of a connector can thus be described *per-state* in a series of *rounds*. *Data flow* on a primitive’s end occurs when a single datum is passed through that end. Within any round data flow may occur on some number of ends. This is equated with the notion of *synchrony*. Hence, the semantics of a connector can be defined in terms of two kinds of constraints:

Synchronisation constraints describe the sets of ends that can be synchronised in a particular step. For example, synchronous channel types permit data flow either on both of their ends or on neither end, and some asynchronous channel types permit data flow on at most one of their two ends.

Data flow constraints describe the data flowing on the ends that have synchronised. For example, such a constraint may say that the data item flowing on the source end of a synchronous channel is the same as the data item flowing on its sink end. It may say how the data item flowing from a source to a sink end is transformed, or that there is no constraint on the data flow, such as for drain channel types which simply discard their data. It may say that the data satisfies a particular predicate, as is the case for filter channels.

We now give an informal description of some of the most commonly used Reo primitives. Note that for all of these primitives, no data flow at all, i.e., doing nothing, is one of its behavioural possibilities.

⁴Available from <http://reo.project.cwi.nl/>

Replicator $\left(a \begin{array}{l} \nearrow b \\ \searrow c \end{array} \right)$ replicates data synchronously from a to b and c .

Thus, data flow either at all ends or not anywhere, and the value of the data at the ends b and c is the value at the end a . An n -replicator behaves similarly, replicating data synchronously from the source end to all the n sink ends.

Merger $\left(\begin{array}{l} a \\ b \end{array} \rightarrow c \right)$ copies data synchronously from a or b to c , but not

from both. Thus data flow on the ends a and c (and not on the end b) or on the ends b and c (and not on the end a), where the value of the data is equal on both ends where data flow.

LossySync $\left(a \dashrightarrow b \right)$ has two possible behaviours with data flow. Data can either flow synchronously from a to b , when possible, or can flow on a but not on b , in which case the value flowing in a is irrelevant.

SyncDrain $\left(a \rightleftarrows b \right)$ acts purely to synchronise ends a and b , thus data flow at end a if and only if they flow at the end b , and there is no constraint on the values of the data.

FIFO₁ $\left(a \text{---} \square \rightarrow b \right)$ has two possible states: empty or full. When the state is empty it can receive a data item from a , changing its state to full. When the state is full it cannot receive any more data, but it can send the previously received data item through b , changing its state to back to empty.

The behaviour of connectors was originally described in terms of data flow through the channels and the nodes connecting them, along with the synchronisation and mutual exclusion constraints they impose. Components attached at the boundary of a connector either attempt to write data to or read data from the end of a channel. The connector coordinates the components by determining when the writes and takes succeed, generally by synchronising a collection of such actions. Data flows from an output end of a primitive to the input end to which it is connected, thus synchronising the two ends. A primitive determines whether data is accepted on an input end (or offered on an output end) based on data flow on its other ends, its state, and any non-deterministic choices it can make. In principle, data continues to flow like this through the connector, with primitives routing the data based on their internal behavioural constraints and the possibilities offered by the surrounding context. Primitives act to locally synchronise actions or to exclude the possibility of actions occurring synchronously. These ‘constraints’ are propagated through the connector, under the restriction that the only communication between entities occurs through the channels. Consequently, the behaviour depends upon the combined choices of primitives and what possibilities the components offer, all of which are not known locally to primitives.

For the purposes of this paper, we do not distinguish between primitives such as channels used for coordination and the components being coordinated, in that both will offer constraints describing their behavioural possibilities. The main difference between primitives and components is that the connector has more control over and more knowledge about the possible behaviour of the former than the latter, though this distinction is blurred in the present framework.

Connectors are formed by plugging the ends of primitives together in a 1:1 fashion, connecting a sink end to a source end, to form *nodes*. Data flows through a connector from primitive to primitive through nodes, subject to the constraint that nodes cannot buffer data. This means that the two ends in a node are synchronised and have the same data flow. When we subsequently encode nodes as constraints, we can simply use the same name for the sink and source ends of the node, because the both the synchronisation and the data flow for these two ends are always identical.

The following examples illustrate *Reo*'s semantics, using the primitives introduced above. We start with a simple example of an exclusive router, after which we present a more complex example that coordinates the control flow of two components.

Example 1. *The connector in Figure 1 is an exclusive router built by composing two LossySync channels (b-e and d-g), one SyncDrain (c-f), one Merger (h-i-f), and three Replicators (a-b-c-d, e-j-h and g-i-k).*

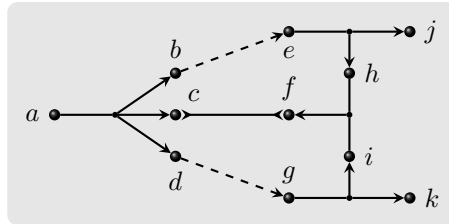


Figure 1: Exclusive router connector.

The constraints of these primitives can be combined to give the following two behavioural possibilities (plus the no flow everywhere possibility):

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

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

We now introduce some special notation. The symbol (\otimes) denotes an exclusive router, as described in Example 1. An exclusive router can easily be

generalised to have any number of sink ends. Nodes (\bullet) can also have multiple source and sink ends to denote generalised mergers and replicators.

The next example illustrates a more complex coordination pattern. Only an informal description of the behaviour of this connector is given, to provide a better intuition of the expressive power of $\mathcal{R}eo$.

Example 2. *The synchronous merge connector, presented in Figure 2, is one of the workflow patterns defined by Van der Aalst [49]. The connector controls the execution of two components A and B such that either A executes, or B executes, or both execute and the connector synchronises on the completion of A and B. The components A and B are represented by boxes with a source end on their left side and a sink end on their right side. We assume that these components receive a signal that triggers their execution via their source end, and that they return a signal on their sink end after their execution is complete.*

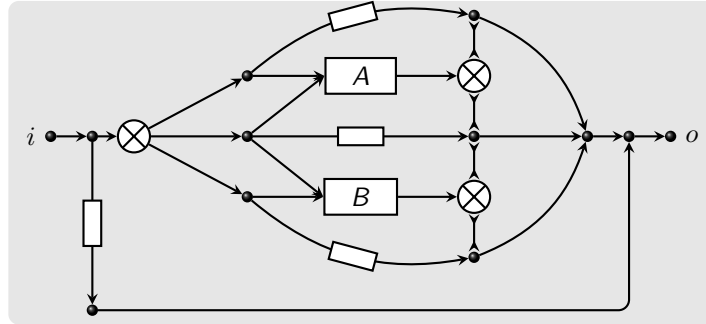


Figure 2: Synchronous merge connector.

One of the main differences with the exclusive router connector is that the synchronous merge connector contains stateful channels, namely $FIFO_1$ channels. The behaviour of the synchronous merge connector depends on the state of its $FIFO_1$ buffers. The expected behaviour, regarding the flow of data on ends i , o , and on the ends of A and B , is as follows. Initially only the source end i can have data flowing, causing one or both of the components to start executing, and changing the state of two of the $FIFO_1$ buffers to full. After this, the only possible behaviour is to wait for the components that started to execute to finish and output a signal. In the same step the sink end will have data flow, and the two $FIFO_1$ buffers will become empty. The $FIFO_1$ channel connected to i prevents new requests which would start A and/or B from being accepted before the previous call to these components is finished. Better intuition for this connector is given by an animation that can be found online.⁵

Section 3 contains a formal definition of the primitives used in these exam-

⁵<http://www.cwi.nl/~proenca/webreo/generated/syncmerge/frameset.htm>

ples, and verification that the composition of the primitives in the exclusive router connector yields the expected behaviour.

In this paper, we address the challenge of implementing $\mathcal{R}eo$ by adopting the view of a $\mathcal{R}eo$ connector as a set of constraints, based on the way the primitives are connected together, and their current state, governing the possible synchronisation and data flow at the channel ends connected to external entities.

3. Coordination via Constraint Satisfaction

In this section we formalise the per-round semantics of $\mathcal{R}eo$ primitives and their composition as a set of constraints. The possible coordination patterns can then be determined using traditional constraint satisfaction techniques.

The constraint-based approach to $\mathcal{R}eo$ is developed in four phases:

synchronisation and data flow constraints describe synchronisation and the data flow possibilities for a single step;

state constraints incorporate next state behaviour into the constraints, enabling the complete description of behaviour in terms of constraints;

external constraints capture externally maintained state, and externally specified transformations and predicates, in order to model a wider selection of primitives and the external entities coordinated by $\mathcal{R}eo$; and whose behaviour changes non-monotonically with the context in which they are placed.

The resulting model significantly extends $\mathcal{R}eo$ implementations with data-aware and context dependent behaviour, and enables more efficient implementation techniques.

3.1. Mathematical Preliminaries

Let \mathcal{X} be the set of ends in a connector. Let $\hat{\mathcal{X}}$ denote the set of variables of set \mathcal{X} annotated with a little hat. Let $Data$ be the domain of data, and define $Data_{\perp} \hat{=} Data \cup \{\text{NO-FLOW}\}$, where $\text{NO-FLOW} \notin Data$ represents ‘no data flow.’ Constraints are expressed in quantifier-free, first-order logic over two kinds of variables: *synchronisation variables* $x \in \mathcal{X}$, which are propositional (boolean) variables, and *data flow variables* $\hat{x} \in \hat{\mathcal{X}}$, which are variables over $Data_{\perp}$. Constraints are formulæ in the following grammar:

$$\begin{aligned} t &::= \hat{x} \mid d && \text{(terms)} \\ a &::= x \mid P(t_1, \dots, t_n) && \text{(atoms)} \\ \psi &::= a \mid \top \mid \psi \wedge \psi \mid \neg\psi && \text{(formulæ)} \end{aligned}$$

where $d \in Data_{\perp}$ is a data item, \top is *true*, and P is an n -arity predicate over terms. One such predicate is equality, which is denoted using the standard infix notation $t_1 = t_2$. The other logical connectives can be encoded as usual:

$\perp \hat{=} \neg \top$; $\psi_1 \vee \psi_2 \hat{=} \neg(\neg\psi_1 \wedge \neg\psi_2)$; $\psi_1 \rightarrow \psi_2 \hat{=} \neg\psi_1 \vee \psi_2$; and $\psi_1 \leftrightarrow \psi_2 \hat{=} (\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1)$.

A *solution* to a formula ψ defined over ends \mathcal{X} is a pair of assignments of types $\sigma : \mathcal{X} \rightarrow \{\perp, \top\}$ and $\delta : \widehat{\mathcal{X}} \rightarrow \mathit{Data}_\perp$, such that σ and δ satisfy ψ , according to the satisfaction relation $\sigma, \delta \models \psi$, defined as follows:

$$\begin{array}{ll} \sigma, \delta \models \top & \text{always} \\ \sigma, \delta \models x & \text{iff } \sigma(x) = \top \\ \sigma, \delta \models P(t_1, \dots, t_n) & \text{iff } (Val_\delta(t_1), \dots, Val_\delta(t_n)) \in \mathcal{I}(P) \end{array} \quad \begin{array}{ll} \sigma, \delta \models \psi_1 \wedge \psi_2 & \text{iff } \sigma, \delta \models \psi_1 \text{ and } \sigma, \delta \models \psi_2 \\ \sigma, \delta \models \neg\psi & \text{iff } \sigma, \delta \not\models \psi \end{array}$$

Each n -ary predicate symbol P has an associated interpretation, denoted by $\mathcal{I}(P)$, such that $\mathcal{I}(P) \subseteq \mathit{Data}_\perp^n$. The function $Val_\delta(t)$ performs a substitution on term t replacing each variable \hat{x} in t by $\delta(\hat{x})$.

The logic with constraints over boolean variables, plus equality constraints over data flow variables, arbitrary terms (beyond the flat Data domain), and top-level existential quantifiers is decidable and in NP [50].

3.2. The Flow Axiom

The NO-FLOW value is used in constraints as a special value when no data flow occurs. A synchronisation variable set to \perp plays the exact same role. These two facts are linked by the following constraint, which links synchronisation and data flow by capturing the relationship between no flow on end $x \in \mathcal{X}$ and the value NO-FLOW:

$$\neg x \leftrightarrow \hat{x} = \text{NO-FLOW} \quad (\text{flow axiom})$$

This axiom applies to all ends in the connector. Let $\mathbf{Flow}(\mathcal{X})$ denote $\bigwedge_{x \in \mathcal{X}} (\neg x \leftrightarrow \hat{x} = \text{NO-FLOW})$. A solution to a set of constraints that satisfies $\mathbf{Flow}(\mathcal{X})$ is called a *firing*. Since we are exclusively interested in finding firings (as opposed to other solutions), we assume that the flow axiom holds for all ends involved.

3.3. Encoding Primitives as Constraints

Two kinds of constraints describe connector behaviour: *synchronisation constraints* (SC) and *data flow constraints* (DFC). The former are constraints over a set \mathcal{X} of boolean variables, describing the presence or absence of data flow at each end—that is, whether or not those ends synchronise. The latter constraints involve in addition data flow variables from $\widehat{\mathcal{X}}$ to describe the data flow at the ends that synchronise.

Table 1 presents the semantics of some commonly used channels and other primitives in terms of synchronisation constraints and data flow constraints. All primitives are stateless apart from the FIFO_1 buffer, which can either be in state FIFOEmpty_1 when it is empty, or in state $\text{FIFOFull}_1(d)$ when it is full with data d . Note that infinite states can exist when the data domain is infinite—a complete account of stateful connectors is given in Section 4 where state handling is introduced into constraints.

Channel	Representation	SC	DFC
Sync	$a \longrightarrow b$	$a \leftrightarrow b$	$\widehat{a} = \widehat{b}$
SyncDrain	$a \longleftarrow b$	$a \leftrightarrow b$	\top
SyncSpout	$a \longleftrightarrow b$	$a \leftrightarrow b$	\top
AsyncDrain	$a \dashrightarrow b$	$\neg(a \wedge b)$	\top
AsyncSpout	$a \dashleftarrow b$	$\neg(a \wedge b)$	\top
LossySync	$a \dashrightarrow b$	$b \rightarrow a$	$b \rightarrow (\widehat{a} = \widehat{b})$
Merger	$\begin{array}{c} a \\ b \end{array} \rightarrow c$	$(c \leftrightarrow (a \vee b)) \wedge \neg(a \wedge b)$	$a \rightarrow (\widehat{c} = \widehat{a}) \wedge b \rightarrow (\widehat{c} = \widehat{b})$
Replicator	$a \rightarrow \begin{array}{c} b \\ c \end{array}$	$(a \leftrightarrow b) \wedge (a \leftrightarrow c)$	$\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a}$
3-Replicator	$a \rightarrow \begin{array}{c} b \\ c \\ d \end{array}$	$(a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (a \leftrightarrow d)$	$\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a} \wedge \widehat{d} = \widehat{a}$
FIFOEmpty ₁	$a \xrightarrow{\square} b$	$\neg b$	\top
FIFOFull ₁ (d)	$a \xrightarrow{\square d} b$	$\neg a$	$b \rightarrow (\widehat{b} = d)$
Filter(P)	$a \xrightarrow{P} b$	$b \rightarrow a$	$b \rightarrow (P(\widehat{a}) \wedge \widehat{a} = \widehat{b}) \wedge (a \wedge P(\widehat{a})) \rightarrow b$

Table 1: Channel Encodings

Sync, SyncDrain and SyncSpout channels Synchronous channels allow data flow to occur only synchronously at both channel ends. SyncSpouts can be viewed as data generators. A more refined variant uses predicates P and Q to constrain the data produced, with data flow constraint such as $a \rightarrow (P(\widehat{a}) \wedge Q(\widehat{b}))$.

AsyncDrain and AsyncSpout These asynchronous channels allow flow on at most one of their two ends. A refined variant of the AsyncSpout has data flow constraint $a \rightarrow P(\widehat{a}) \wedge b \rightarrow Q(\widehat{b})$.

Non-deterministic LossySync A LossySync can always allow data flow on end a . It can in addition, non-deterministically, allow data flow on end b , in which case the data from a is passed to b . Note that the LossySync was initially intended to be context-dependent, i.e., to lose data only when b could not accept data, but semantic models such as the constraint automata were insufficient to describe it. We return to context dependency in Section 5.

FIFOEmpty₁ and FIFOFull₁(d) FIFO₁ is a stateful channel representing a buffer of size 1. When the buffer is empty it can only receive data on a ,

but never output data on b . When it is full with data d , it can only output d through b , but cannot receive more data on a .

Merger A merger permits data flow synchronously through one of its source ends, exclusively, to its sink end.

Replicator A replicator (and a 3-replicator) allows data to flow only synchronously at every channel end. Data is replicated from the source end to every sink end. The constraints for the n -replicator (such as the 3-replicator found in Example 1) can be easily derived based on the constraints for the replicator, as show above.

Filter A filter permits data matching its filter predicate $P(\hat{x})$ to pass through synchronously, otherwise the data is discarded.

In our approach, a channel or primitive can have any behaviour that can be specified in the language of the underlying constraint solver, together with the flow axiom, so long as the no-flow solution, i.e., an assignment that maps each synchronous variable to false, is admitted by the constraints. We do not impose any additional conditions in this paper. Note that this precludes channels which, for example, profess to offer quality of service guarantees.

Curiously, the constraints for some channels, such as SyncDrain and SyncSpout, are identical, indicating that the model does not strongly account for the direction of data flow. Typically, however, some variables will be bound to a value and others will remain unbound, and data can be seen as flowing from the bound variables to the unbound ones. In \mathcal{Reo} , the direction of the data flow is used to govern the well-formedness of connector composition, so that connectors have the expected semantics, but our constraints ignore it. Our constraints will be solved classically, in contrast to the intuitionistic model of Clarke [18], which was designed to avoid causality problems resulting from taking into account the direction of data flow. In our setting the direction of data flow can still be used to optimise the constraint solving, as it is generally more efficient to start with constrained variables than with unconstrained variables, but we do not explore optimisations beyond those provided in the constraint solver underlying in our prototype implementation.

Other channels can use non-trivial predicates over more than one argument. For example, it is possible to define a special synchronous drain variant whose predicate $P(\hat{a}, \hat{b})$ constrains the data on both of its ends, for instance, by requiring that they are equal or that the content of a field containing the geographic location corresponding to the data on \hat{a} is nearby the location of \hat{b} . The data flow constraints of this variation of the synchronous drain can be defined, for example, as $SameLocation(\hat{a}.location, \hat{b}.location)$, assuming that $-.location$ extracts the location field from the data and the predicate $SameLocation$ determines whether two locations are the same or not.

Splitting the constraints into synchronisation and data flow constraints is very natural, and it closely resembles the constraint automata model [14] (see Section 4.4). It also enables some implementation optimisations, if we require

that the synchronisation constraints ψ_{SC} are an abstraction of the overall constraints ψ , i.e., requiring that $\sigma, \delta \models \psi$ implies $\sigma \models \psi_{SC}$, for any solution pair (σ, δ) . Following Sheini and Sakallah [48], for example, a SAT solver can be applied to the synchronisation constraints, efficiently ruling out many non-solutions. In many cases, a solution to the synchronisation constraints actually guarantees that a solution to the data flow constraints exists. The only primitive in Table 1 for which this is not true is the filter, as it inspects the data in order to determine its synchronisation constraints.

3.4. Combining connectors

Two connectors can be plugged together whenever for each end x appearing in both connectors, x is only a sink end in one connector and only a source end in the other.⁶ If the constraints for the two connectors are ψ_1 and ψ_2 , then the constraints for their composition is simply $\psi_1 \wedge \psi_2$. Existential quantification, such as $\exists x. \exists \hat{x}. \psi$, can be used to abstract away intermediate channel ends (such as x).

Top-level constraints are given by the following grammar:

$$\mathcal{C} ::= \psi \mid \mathcal{C} \wedge \mathcal{C} \mid \exists x. \mathcal{C} \mid \exists \hat{x}. \mathcal{C} \quad (\text{top-level constraints})$$

where ψ is as defined in Section 3.1. Top-level constraints are used to introduce the existential quantifier, which hides the names of internal ends after composition is performed. This opens new possibilities for optimisation by ignoring ends that are not relevant for the final solution. This optimisation can be done statically, but our engine implementations do not address this, beyond what is already handled in the SAT and constraint solvers.

The satisfaction relation is extended with the cases:

$$\begin{aligned} \sigma, \delta \models \exists x. \mathcal{C} & \quad \text{iff} \quad \text{there exists a } b \in \{\top, \perp\} \text{ such that } \sigma, \delta \models \mathcal{C}[b/x] \\ \sigma, \delta \models \exists \hat{x}. \mathcal{C} & \quad \text{iff} \quad \text{there exists a } d \in \text{Data}_\perp \text{ such that } \sigma, \delta \models \mathcal{C}[d/\hat{x}]. \end{aligned}$$

$\mathcal{C}[a/x]$ is the constraint resulting from replacing all free occurrences of x by a in \mathcal{C} , in the usual fashion.

The following constraints describe the composition of the primitives for the connector presented in Example 1, abstracting away the internal ends:

$$\begin{aligned} \Psi_{SC} &= (a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (a \leftrightarrow d) \wedge (e \rightarrow b) \wedge (c \leftrightarrow f) \wedge (g \rightarrow d) \wedge (e \leftrightarrow j) \\ &\quad \wedge (e \leftrightarrow h) \wedge (f \leftrightarrow (h \vee i)) \wedge \neg(h \wedge i) \wedge (g \leftrightarrow i) \wedge (g \leftrightarrow k) \\ \Psi_{DFC} &= (a \rightarrow (\hat{b} = \hat{a} \wedge \hat{c} = \hat{a})) \wedge (e \rightarrow \hat{b} = \hat{e}) \wedge (g \rightarrow \hat{d} = \hat{g}) \wedge \hat{j} = \hat{e} \wedge \hat{h} = \hat{e} \wedge \\ &\quad (h \rightarrow \hat{f} = \hat{h}) \wedge (i \rightarrow \hat{f} = \hat{i}) \wedge \hat{i} = \hat{g} \wedge \hat{k} = \hat{g} \\ \Psi &= \exists \mathcal{X}. \exists \hat{\mathcal{X}}. (\Psi_{SC} \wedge \Psi_{DFC} \wedge \text{Flow}(\mathcal{X} \cup \{a, j, k\})) \\ &\quad \text{where } \mathcal{X} = \{b, c, d, e, f, g, h, i\} \end{aligned}$$

⁶The information about whether an end is a sink or source needs to be maintained at a level above the constraints. Incorporating such information into the constraints would be straightforward.

A SAT solver can quickly solve the constraint Ψ_{SC} (ignoring internal ends):⁷

$$\sigma_1 = a \wedge j \wedge \neg k \quad \sigma_2 = a \wedge \neg j \wedge k \quad \sigma_3 = \neg a \wedge \neg j \wedge \neg k$$

Using these solutions, Ψ can be simplified using standard techniques as follows:

$$\begin{aligned} \Psi \wedge \sigma_1 &\rightsquigarrow \widehat{j} = \widehat{a} \wedge \widehat{k} = \text{NO-FLOW} \\ \Psi \wedge \sigma_2 &\rightsquigarrow \widehat{k} = \widehat{a} \wedge \widehat{j} = \text{NO-FLOW} \\ \Psi \wedge \sigma_3 &\rightsquigarrow \widehat{a} = \text{NO-FLOW} \wedge \widehat{j} = \text{NO-FLOW} \wedge \widehat{k} = \text{NO-FLOW} \end{aligned}$$

These solutions say that data can flow either from end a to j , or from end a to k , or that no flow is possible in any of the ends, as expected.

3.5. Refinement and Behavioural Equivalence

The notions of refinement and behavioural equivalence for connectors, as they have been presented thus far, are straightforward. Connectors, viewed as constraints, can be compared by using their set of possible solutions. Let ψ_1 and ψ_2 be the constraints for connectors \mathcal{C}_1 and \mathcal{C}_2 , respectively. We say ψ_1 is *refined* by ψ_2 , denoted by $\psi_1 \leq \psi_2$, if the set of solutions for ψ_1 contains the set of solutions for ψ_2 , i.e.,

$$\psi_1 \leq \psi_2 \quad \hat{=} \quad \{\sigma, \delta \mid \sigma, \delta \models \psi_1\} \supseteq \{\sigma, \delta \mid \sigma, \delta \models \psi_2\}$$

More simply, we can say $\psi_1 \leq \psi_2$ if and only if $\psi_1 \rightarrow \psi_2$ is true. Furthermore, two constraints ψ_1 and ψ_2 are *behavioural equivalent* if and only if $\psi_1 \leq \psi_2$ and $\psi_2 \leq \psi_1$, that is, if they are logically equivalent.

In the rest of this paper we will extend this constraint-based model to describe the evolution of constraints over time (adding state information) and context dependency. The notions of refinement and behavioural equivalence are not dramatically affected by these extensions. We will redress refinement and behavioural equivalence in Section 4.3.

4. Adding State

Adding constraints to capture stateful primitives is relatively easy. This involves, firstly, adding constraints to capture the pre- and post-states for each interaction described by the constraints, and, secondly, providing details of how constraints change to represent the changes in state selected by the $\mathcal{R}eo$ engine. In addition, we show the correspondence between our encoding and the semantics of $\mathcal{R}eo$ in terms of constraint automata (CA).

⁷We use the open source SAT solver: <http://www.sat4j.org/>.

4.1. Encoding State Machines

Primitives such as the FIFO_1 channel are stateful, i.e., their state and subsequent behaviour change after data flow through the channel. This is exemplified in the constraint automata (CA) semantics of $\mathcal{R}\text{eo}$ [14]. The CA of a FIFO_1 channel is shown in Figure 3. Its initial state is **empty**. From this state the automaton can take a transition to state $\text{full}(d)$ if there is data flow on end a , excluding data flow on end b . The constraint $d = \widehat{a}$ models the storing of the value flowing on end a into the internal state variable d . The transition from $\text{full}(d)$ to **empty** is read in a similar way, except that the data is moved from the internal state variable d to end b .

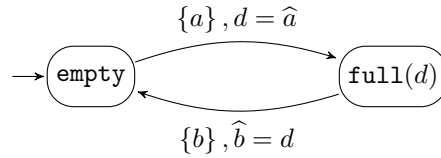


Figure 3: Constraint Automata for the FIFO_1 channel

To encode state information, the logic is extended so that terms also include n -ary uninterpreted function symbols:

$$t ::= \widehat{x} \mid f(t_1, \dots, t_n) \quad (\text{terms})$$

A term t is *ground* iff $t = f(t_1, \dots, t_n)$ and each t_i for $1 \leq i \leq n$ is ground. Thus a term containing a variable is not ground. The set Data described earlier can now be seen as ground 0-ary uninterpreted function symbols; we now consider Data to be the Herbrand universe over the set on uninterpreted function symbols.

Let S be the set of stateful primitives in a connector. Add a new set of term variables $state_p$ and $state'_p$, for each $p \in S$, to denote the state before and after the present step. State machines of primitives are encoded by encoding their constraint automata [32]. In Section 4.4 we present the correctness and the compositionality of our encoding with respect to constraint automata. For example, the state machine of a FIFO_1 channel is encoded as the formula:

$$\begin{aligned} & state_{\text{FIFO}_1} = \text{empty} \rightarrow \left\{ \begin{array}{l} \neg b \wedge \\ a \rightarrow (state'_{\text{FIFO}_1} = \text{full}(\widehat{a})) \wedge \\ \neg a \rightarrow (state'_{\text{FIFO}_1} = state_{\text{FIFO}_1}) \end{array} \right. \wedge \left\{ \begin{array}{l} state_{\text{FIFO}_1} = \text{full}(d) \rightarrow \\ \neg a \wedge \\ b \rightarrow (\widehat{b} = d \wedge state'_{\text{FIFO}_1} = \text{empty}) \wedge \\ \neg b \rightarrow (state'_{\text{FIFO}_1} = state_{\text{FIFO}_1}) \end{array} \right. \\ & \wedge \quad \neg a \wedge \neg b \rightarrow state'_{\text{FIFO}_1} = state_{\text{FIFO}_1} \end{aligned}$$

The final conjunct captures when no transition occurs due to no data flow.

To complete the encoding, we add a formula describing the present state to the mix, i.e., we add conjunctively a formula that defines the value of $state_p$

for each stateful primitive p , for the current state. In our example, the formula $state_{\text{FIFO}_1} = \text{empty}$ records the fact that the FIFO_1 is in the empty state, whereas $state_{\text{FIFO}_1} = \text{full}(d)$ records the fact that it is in the full state, containing data d .

In general, the state of primitives will be encoded as a formula of the form $\bigwedge_{p \in S} state_p = t_p$. This is called a *pre-state vector*. Similarly, $\bigwedge_{p \in S} state'_p = t_p$, is called the *post-state vector*. The pre-state vector describes the state of the connector before constraint satisfaction; the post-state vector describes the state after constraint satisfaction, that is, it gives the next state.

Note that stateless primitives do not contribute to the state vector.

4.2. A Constraint Satisfaction-based Engine for $\mathcal{R}eo$

Constraint satisfaction techniques can now form the working heart of an implementation of an *engine* performing the coordination described in $\mathcal{R}eo$ connectors. The engine holds the current set of constraints, called a *configuration*, and operates in *rounds*, each of which consists of a *solve* step, which produces a solution for the constraints, and an *update* step, which uses the solution to update the constraints to model the transition to a new state. This is depicted in the diagram in Figure 4.

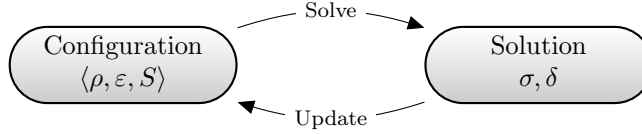


Figure 4: Phases of the $\mathcal{R}eo$ Engine

The *configuration* of the engine is a triple $\langle \rho, \varepsilon, S \rangle$, where ρ represents *persistent* constraints, ε represents *ephemeral* constraints, and S is the set of stateful primitives in the connector. The persistent constraints are eternally true for a connector, including constraints such as the description of the state machines of the primitives. The ephemeral constraints include the encoding of the pre-state vector. These constraints are updated each round. A full round can be represented as follows, where the superscript indicates the round number:

$$\langle \rho, \varepsilon^n, S \rangle \xrightarrow{\text{solve}} \langle \sigma^n, \delta^n \rangle \xrightarrow{\text{update}} \langle \rho, \varepsilon^{n+1}, S \rangle$$

satisfying:

$$\sigma^n, \delta^n \models \rho \wedge \varepsilon^n \quad (\text{solve})$$

$$\varepsilon^{n+1} \equiv \bigwedge_{p \in S} state_p = \delta^n(state'_p) \quad (\text{update})$$

We write $\langle \rho, \varepsilon, S \rangle \xrightarrow{\sigma, \delta} \langle \rho, \varepsilon', S \rangle$ to denote that $\sigma, \delta \models \rho \wedge \varepsilon$ and $\varepsilon' = \bigwedge_{p \in S} state_p = \delta(state'_p)$. Furthermore, we write $c \rightarrow c'$ when there is a pair σ, δ

such that $c \xrightarrow{\sigma, \delta} c'$, and we write \rightarrow^* to denote the reflexive and transitive closure of \rightarrow . We now provide a more formal account of our encoding of constraint automata into constraints.

4.3. Refinement and Behavioural Equivalence Revisited

Previously we defined refinement and behavioural equivalence for a single step, disregarding state updates. We now outline the extension to refinement and behavioural equivalence to take care of these features. The notion of refinement parallels the standard notion of simulation in transition systems.

A connector in configuration $c_1 = \langle \rho_1, \varepsilon_1, S_1 \rangle$ is refined by another connector in configuration $c_2 = \langle \rho_2, \varepsilon_2, S_2 \rangle$, denoted as $c_1 \leq c_2$, if and only if:

1. $\rho_1 \wedge \varepsilon_1 \leq \rho_2 \wedge \varepsilon_2$, and
2. for every σ and δ such that $c_1 \xrightarrow{\sigma, \delta} c'_1$, it holds that $c_2 \xrightarrow{\sigma, \delta} c'_2$ and $c'_1 \leq c'_2$.

We define the behavioural equivalence relation, denoted by the operator \approx , as the greatest relation on pairs of configurations c_1 and c_2 , such that $c_1 \leq c_2$ and $c_2 \leq c_1$, which correspond to the pairs that obey the standard back and forth conditions of bisimulation with respect to our definition of refinement.

We now argue that this notion of equivalence is a congruence. In our setting, a context Γ consists of a disjoint set of primitives composed by conjunctively adding constraints. We define Γ as a configuration $\langle \rho_\Gamma, \varepsilon_\Gamma, S_\Gamma \rangle$, and we define the application of a context Γ to a configuration $c = \langle \rho, \varepsilon, S \rangle$ as $\Gamma(c) = \langle \rho \wedge \rho_\Gamma, \varepsilon \wedge \varepsilon_\Gamma, S \uplus S_\Gamma \rangle$, where \uplus denotes the disjoint union of sets. Observe that \approx is a congruence relation, that is, for any context Γ ,

$$c_1 \approx c_2 \quad \Rightarrow \quad \Gamma(c_1) \approx \Gamma(c_2). \quad (1)$$

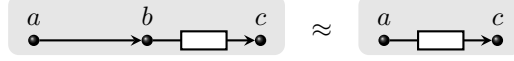
We now sketch the proof of Equation 1. Construct a relation $R = \{(c'_1, c'_2) \mid (\Gamma(c_1), \Gamma(c_2)) \rightarrow^* (c'_1, c'_2)\}$, where $(c_1, c_2) \xrightarrow{\sigma, \delta} (c'_1, c'_2)$ whenever $c_1 \xrightarrow{\sigma, \delta} c'_1$ and $c_2 \xrightarrow{\sigma, \delta} c'_2$. Trivially, $(\Gamma(c_1), \Gamma(c_2)) \in R$. To show that $\Gamma(c_1) \approx \Gamma(c_2)$ we need to prove that R is a bisimulation. The main idea for this part of the proof is that if $\Gamma(c_1) \xrightarrow{\sigma, \delta} c'_1$, then also $\Gamma(c_2) \xrightarrow{\sigma, \delta} c'_2$ and $(c'_1, c'_2) \in R$, because $c_1 \approx c_2$, and the context is adding new constraints conjunctively. Using an inductive argument we can then show that, for any element (c'_1, c'_2) such that $(\Gamma(c_1), \Gamma(c_2)) \rightarrow^* (c'_1, c'_2)$, also $(c'_1, c'_2) \in R$.

A connector can be seen as a state machine, where each state is defined by the constraint $\rho \wedge \varepsilon$, and the transition relation between states is derived from the valid solutions for the state variables which update the state according to condition 2. Using this analogy, verifying whether a connector in a given state (configuration) is refined by another connector in some other state is equivalent to searching for a simulation in this state machine with respect to the definition of refinement of constraints introduced in Section 3.5.

Note that two connectors can be related by the refinement relation even when considering an infinite data domain and an infinite number of possible solutions

as a result. Similar reasoning is also performed when searching (possibly infinite) bisimulations of coalgebraic system models [45].

Example 3. *In this example we show that:*



Let $\text{sync}(a, b)$ be the constraints of the Sync channel for the ends a and b , and $\text{fifo}(a, b)$ be the persistent constraints of the FIFO_1 for the ends a and b , encoding the state transitions using the variables state and state' . We assume the flow axiom is included in the definition of the constraints of the Sync and the FIFO_1 channels. We now show that the initial configurations are equivalent, i.e.,

$$\phi_{\text{empty}} \approx \psi_{\text{empty}}$$

where

$$\begin{aligned} \phi_{\text{empty}} &= \exists b, \widehat{b}. (\text{sync}(a, b) \wedge \text{fifo}(b, c) \wedge \text{state} = \text{empty}) \\ \psi_{\text{empty}} &= \text{fifo}(a, c) \wedge \text{state} = \text{empty}. \end{aligned}$$

We explain why $\phi_{\text{empty}} \leq \psi_{\text{empty}}$; the reverse refinement can be explained using analogous reasoning. It is easy to see that condition 1. holds for the initial configuration, because when ϕ_{empty} is true then ψ_{empty} will also be true. We now look at the possible solutions for ϕ_{empty} . There are two possible scenarios: (1) either σ assigns the variables a and b to true, the variable c to false, and δ assigns \widehat{a} and \widehat{b} to the same data value d and \widehat{c} to **NO-FLOW**, or (2) σ assigns a , b , and c to false, and δ assigns the corresponding data values to **NO-FLOW**. Let σ_1 and δ_1 be a solution for the first scenario, and σ_2 and δ_2 be the solution for the second scenario. To verify condition 2. we need to update the constraints ϕ_{empty} and ψ_{empty} using the possible solutions of ϕ_{empty} , and confirm that the two conditions also hold for the resulting configurations. δ_2 leads to the same constraints ϕ_{empty} and ψ_{empty} . δ_1 leads ϕ_{empty} and ψ_{empty} to configurations with constraints $\phi_{\text{full}(d)}$ and $\psi_{\text{full}(d)}$, where:

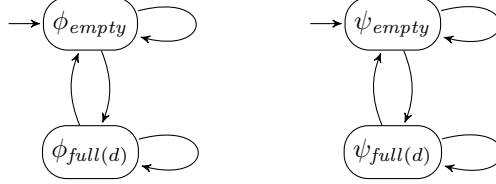
$$\begin{aligned} \phi_{\text{full}(d)} &= \exists b, \widehat{b}. (\text{sync}(a, b) \wedge \text{fifo}(b, c) \wedge \text{state} = \text{full}(d)) \\ \psi_{\text{full}(d)} &= \text{fifo}(a, c) \wedge \text{state} = \text{full}(d). \end{aligned}$$

Using the same reasoning as before, we can show that the possible solutions consist of having no data flow, or of having the data d flowing on the end c . According to condition 2., the first scenario does not change the configuration of the connector, while the second scenario results in the initial configuration, which we already considered before.

Ultimately, we have argued that $\phi_{\text{empty}} \approx \psi_{\text{empty}}$ corresponds to bisimulation

$$R = \{(\phi_{\text{empty}}, \psi_{\text{empty}})\} \cup \{(\phi_{\text{full}(d)}, \psi_{\text{full}(d)}) \mid d \in \text{Data}\}$$

between the following transition systems:



Here each arrow represents all possible solutions that can be applied (according to condition 2) yielding a new configuration.

4.4. Correctness via Constraint Automata

In this section we address the correctness of our approach. We base our argument on the constraint automata model of $\mathcal{R}eo$ [14]. In the process, we make the encoding described above completely formal.

4.4.1. Constraint Automata

We recall the formal definition of constraint automata. Define $DC_{\hat{\mathcal{X}}}$ to be the set of constraints in our language above only over variables in the set $\hat{\mathcal{X}}$, where the underlying data domain is \mathcal{Data} . This excludes constraints over synchronisation variables and constraints involving **NO-FLOW**.

Definition 1 (Constraint Automaton [14]). A constraint automaton (over data domain \mathcal{Data}) is a tuple $\mathcal{A} = \langle Q, \mathcal{X}, \longrightarrow, Q_0 \rangle$, where Q is a set of states, \mathcal{X} is a finite set of end names, \longrightarrow is a subset of $Q \times 2^{\mathcal{X}} \times DC_{\hat{\mathcal{X}}} \times Q$, called the transition relation of \mathcal{A} , and $Q_0 \subseteq Q$ is the set of initial states. We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. For every transition $q \xrightarrow{N,g} p$, we require that g , the guard, is a $DC_{\hat{\mathcal{X}}}$ -constraint. For every state $q \in Q$, there is a transition $q \xrightarrow{\emptyset, \top} q$.

The meaning of transition the $q \xrightarrow{N,g} p$ is that in state q data flows at the ends in the set N , while excluding flow at ends in $\mathcal{X} \setminus N$. The data flowing through the ends N satisfies the constraint g , and the resulting state is p . Thus, in constraint automata, synchronisation is described by the set N and data flow is described by the constraint g . The transition $q \xrightarrow{\emptyset, \top} q$ is present for technical reasons, namely, to simplify the definition of the product, below.

In order to define the language accepted by a constraint automaton,⁸ we need to set up some preliminary definitions. Given a set of end names \mathcal{X} , define *data assignments* to be given by the following alphabet: $\Sigma_{\hat{\mathcal{X}}} = (\hat{\mathcal{X}} \rightarrow \mathcal{Data})$, namely the partial finite maps from the data flow variables to the data domain. Define $\delta \models g$, where $\delta : \hat{\mathcal{X}} \rightarrow \mathcal{Data}$ and data constraint $g \in DC_{\hat{\mathcal{X}}}$, as $\emptyset, \delta \models g$ (from Section 3.1). Observe that this will be well formed, as data constraints $DC_{\hat{\mathcal{X}}}$ do not mention synchronisation variables. Automata will accept a finite words from the set $\Sigma_{\hat{\mathcal{X}}}^*$. As automata have no final states specified, we assume

where:

$$\begin{aligned} Q &= \{\mathbf{empty}\} \cup \{\mathbf{full}(d) \mid d \in \mathcal{Data}\} & \mathcal{X} &= \{a, b\} \\ \longrightarrow &= \{(\mathbf{empty}, \{a\}, \widehat{a} = d, \mathbf{full}(d)) \mid d \in \mathcal{Data}\} & Q_0 &= \mathbf{empty}. \\ &\cup \{(\mathbf{full}(d), \{b\}, \widehat{b} = d, \mathbf{empty}) \mid d \in \mathcal{Data}\} \end{aligned}$$

The trivial transition has been omitted.

4.4.2. Encoding Constraint Automata as Constraints

Given a constraint automata $\mathcal{A} = \langle Q, \mathcal{X}, \longrightarrow, Q_0 \rangle$. An obvious correspondence exists between subsets N of \mathcal{X} and functions $\mathcal{X} \rightarrow \{\perp, \top\}$. Define $\chi_N : \mathcal{X} \rightarrow \{\perp, \top\}$ such that $\chi_N(x) = \top$ if and only if $x \in N$. If δ is defined over $\widehat{N} \subseteq \widehat{\mathcal{X}}$, that is, $\delta : \widehat{N} \rightarrow \mathcal{Data}$, define δ^+ to extend δ to map each $x \in \widehat{\mathcal{X}} \setminus \widehat{N}$ to NO-FLOW.

Definition 3 (Encoding of states). Recall that we deal with constraints on a per state basis. The following conditions characterise constraint ψ_q corresponding to state $q \in Q$, where $q \xrightarrow{N_1, g_1} p_1, \dots, q \xrightarrow{N_n, g_n} p_n$ are the transitions in the automaton with source q :

- a) For all $\sigma : \mathcal{X} \rightarrow \{\perp, \top\}$, $\delta : \widehat{\mathcal{X}} \cup \{\text{state}, \text{state}'\} \rightarrow \mathcal{Data}$ and $p \in Q$, such that $\sigma, \delta \models \psi_q$, $\delta(\text{state}) = q$ and $\delta(\text{state}') = p$, there is a transition $q \xrightarrow{N_i, g_i} p$ such that $\sigma = \chi_{N_i}$ and $\delta \models g_i$.
- b) For all transitions $q \xrightarrow{N_i, g_i} p_i$ and for all $\delta : \widehat{N}_i \rightarrow \mathcal{Data}$ such that $\delta \models g_i$, we have that $\chi_{N_i}, \delta^+ \cup \{\text{state} \mapsto q, \text{state}' \mapsto p_i\} \models \psi_q$.

When the conditions (a) and (b) from Definition 3 hold for a constraint ψ_q , we say that ψ_q encodes state q . These two conditions state that for each data flow described by the constraint, there is a transition in the automaton with the same data flow, and vice versa. (Note that if there is only one state, all mention of state variables can be dropped.)

These constraints are put together to describe the entire state machine as follows:

$$\rho_{\mathcal{A}} = \bigwedge_{q_i \in Q} (\text{state} = q_i \rightarrow \psi_{q_i}).$$

Example 6. The constraints for the FIFO_1 buffer presented in Section 4.1 are correct with respect to the constraint automaton in Example 5. We show this only for the initial state, \mathbf{empty} , but it can be easily verified also for the other state.

For the \mathbf{empty} state, the corresponding constraints and their possible solutions are:

$$\begin{aligned} \psi_{\mathbf{empty}} &= \neg b \wedge a \rightarrow (\text{state}' = \mathbf{full}(\widehat{a})) \wedge \neg a \rightarrow (\text{state}' = \text{state}) \\ \sigma_1 &= a \wedge \neg b & \sigma_2 &= \neg a \wedge \neg b \\ \delta_1 &= \text{state}' = \mathbf{full}(\widehat{a}) & \delta_2 &= \text{state}' = \mathbf{empty} \end{aligned}$$

In the constraint automaton, the transitions from the **empty** state are:

$$\begin{array}{ccc} \mathbf{empty} & \xrightarrow{a, d=\widehat{a}} & \mathbf{full}(d) \\ \mathbf{empty} & \xrightarrow{\emptyset, \top} & \mathbf{empty} \end{array}$$

The two conditions that confirm the correctness of the constraint $\psi_{\mathbf{empty}}$ can now be easily verified, after expanding the constraint $state' = \mathbf{full}(\widehat{a})$ to the logically equivalent constraint $\exists d.(\widehat{a} = d \wedge state' = \mathbf{full}(d))$.

4.4.3. Correctness

Our correctness result shows that, given a constraint automaton of a stateful primitive, every step of the automaton corresponds to a solve-update round in our constraint satisfaction-based engine for $\mathcal{R}eo$, and vice-versa. Recall that $\rho_{\mathcal{A}}$ denotes the encoding of the automata \mathcal{A} as a constraint. In the rest of this paper we write $\langle \rho_{\mathcal{A}}, state = q \rangle$ to denote a configuration of the constraint solver, where $\rho_{\mathcal{A}}$ is the persistent constraint, $state = q$ is the ephemeral constraint, and the $state$ denotes the state variable. Note that we omit the S component containing the set of known stateful primitives, which is used to manage interaction with multiple stateful primitives, as we are dealing with only a single automaton.

Theorem 1. *Let \mathcal{A} be a constraint automata and q a state of \mathcal{A} . Then the following holds:*

$$q \xrightarrow{\delta} p \text{ is a } q\text{-step of } \mathcal{A} \quad \text{iff} \\ \langle \rho_{\mathcal{A}}, state = q \rangle \xrightarrow{\text{solve}} \langle \sigma, \delta' \rangle \xrightarrow{\text{update}} \langle \rho_{\mathcal{A}}, state = p \rangle,$$

where $\widehat{N} = \text{dom}(\delta)$, $\sigma = \chi_N$ and $\delta' = \delta^+ \cup \{state \mapsto q, state' \mapsto p\}$.

Proof. Recall Definition 3 which characterises the encoding of a state q as a constraint ψ_q , and the definition of the solve- and the update-arrow, presented in Section 4.2. The arrow $\langle \rho, \epsilon \rangle \xrightarrow{\text{solve}} \langle \sigma, \delta \rangle$ exists if and only if $\sigma, \delta \models \rho \wedge \epsilon$, while the arrow $\langle \sigma, \delta \rangle \xrightarrow{\text{update}} \langle \rho, \epsilon' \rangle$ exists if and only if $\epsilon' \equiv state = \delta(state')$.

- (\Leftarrow) Assume $\langle \rho_{\mathcal{A}}, state = q \rangle \xrightarrow{\text{solve}} \langle \sigma, \delta' \rangle \xrightarrow{\text{update}} \langle \rho_{\mathcal{A}}, state = p \rangle$. The first solve-arrow indicates that $\sigma, \delta' \models \rho_{\mathcal{A}} \wedge state = q$. Note that $\rho_{\mathcal{A}} \wedge state = q \Leftrightarrow \bigwedge_{r \in Q} (state = r \rightarrow \psi_r) \wedge state = q$, which implies ψ_q by modus ponens. Therefore $\sigma, \delta' \models \psi_q$. Define $\delta = \delta' \upharpoonright \widehat{N}$, where \upharpoonright denotes the standard domain restriction of functions. Observe that $\delta' = \delta^+ \cup \{state \mapsto q, state' \mapsto p\}$. Since $\delta'(state) = q$ and $\delta'(state') = p$, it follows from condition (a) in Definition 3 that there is a transition $q \xrightarrow{N, g} p$, where $\sigma = \chi_N$ and $\delta' \models g$. To show that $q \xrightarrow{\delta} p$ we just need to verify that also $\delta \models g$. This follows because g refers only to variables in \widehat{N} , and because $\text{dom}(\delta) = \widehat{N}$.

- (\Rightarrow) Assume $q \xrightarrow{\delta} p$. By the definition of q -step, there is a transition $q \xrightarrow{N,g} p$ from \mathcal{A} such that $\widehat{N} = \text{dom}(\delta)$ and $\delta \models g$. Let $\sigma = \chi_N$ and $\delta' = \delta^+ \cup \{\text{state} \mapsto q, \text{state}' \mapsto p\}$. From condition (b) in Definition 3, it follows that $\sigma, \delta' \models \psi_q$. Observe that, because $\delta'(\text{state}) = q$, we conclude that $\sigma, \delta' \models \text{state} = q$. Hence (1) $\sigma, \delta' \models (\text{state} = q \rightarrow \psi_q) \wedge \text{state} = q$. Furthermore, $\delta'(\text{state}) = q$ also implies that for every state $q' \neq q$, the formula $\text{state} = q'$ does not hold, thus (2) $\sigma, \delta' \models \bigwedge_{r \in Q \setminus \{q\}} (\text{state} = r \rightarrow \psi_r)$. From (1) and (2) we conclude that $\sigma, \delta' \models \bigwedge_{r \in Q} (\text{state} = r \rightarrow \psi_r) \wedge \text{state} = q$. Therefore, by the definition of the solve-arrow, $\langle \rho_{\mathcal{A}}, \text{state} = q \rangle \xrightarrow{\text{solve}} \langle \sigma, \delta' \rangle$. Finally, since $\delta'(\text{state}') = p$, we have by the definition of the update-arrow that, that $\langle \sigma, \delta' \rangle \xrightarrow{\text{update}} \langle \rho_{\mathcal{A}}, \text{state} = p \rangle$.

□

4.4.4. Compositionality

We now argue that the composition of two constraint automata describing two connectors composed appropriately (sink-to-source) corresponds to composition (conjunction) of their corresponding constraints (per state). In both cases, the overlapping of end names corresponds to the places where connectors are joined. Assume that we have constraint automata \mathcal{A}_i with domains \mathcal{X}_i , for $i \in \{1, 2\}$. In constraint automata, composition is defined by the following rule:

$$\frac{q_1 \xrightarrow{N_1, g_1} p_1 \in \mathcal{A}_1 \quad q_2 \xrightarrow{N_2, g_2} p_2 \in \mathcal{A}_2 \quad N_1 \cap \mathcal{C} = N_2 \cap \mathcal{C}}{(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (p_1, p_2) \in \mathcal{A}_1 \bowtie \mathcal{A}_2}$$

where $\mathcal{C} = \mathcal{X}_1 \cap \mathcal{X}_2$ is the set of shared ends of both automata.

Assume that ψ_{q_1} and ψ_{q_2} are the constraints for states q_1 and q_2 of automata \mathcal{A}_1 and \mathcal{A}_2 , respectively, as described above. We claim that $\psi_{q_1} \wedge \psi_{q_2}$ is the constraint modelling state (q_1, q_2) in the composite automaton given by the above rule. Note that we would need to add equations such as $\text{state}_{1 \times 2} = (\text{state}_1, \text{state}_2)$ and $\text{state}'_{1 \times 2} = (\text{state}'_1, \text{state}'_2)$ to make the format of the equations match up—a fact we gloss over, as the exact representation of the internal state is not observable.

Lemma 1. *Assume condition (a) from Definition 3 holds for two constraints ψ_{q_1} and ψ_{q_2} , and for automata \mathcal{A}_1 and \mathcal{A}_2 with domains \mathcal{X}_1 and \mathcal{X}_2 , respectively. Then condition (a) also holds for $\psi_{q_1} \wedge \psi_{q_2}$ and automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$.*

Proof. Let $\sigma \upharpoonright \mathcal{X}$ denote σ restricted to domain \mathcal{X} . Assume that $\sigma, \delta \models \psi_{q_1} \wedge \psi_{q_2}$, where $\text{dom}(\sigma) = \mathcal{X}_1 \cup \mathcal{X}_2$, $\text{dom}(\delta) = \widehat{\mathcal{X}}_1 \cup \widehat{\mathcal{X}}_2 \cup \{\text{state}_1, \text{state}_2, \text{state}'_1, \text{state}'_2\}$, $\delta(\text{state}_i) = q_i$, and $\delta(\text{state}'_i) = p_i$, for $i \in \{1, 2\}$. It follows that we have $\sigma \upharpoonright \mathcal{X}_1, \delta \models \psi_{q_1}$ and $\sigma \upharpoonright \mathcal{X}_2, \delta \models \psi_{q_2}$ and, from the properties of ψ_{q_i} , that there exists a transition $q_i \xrightarrow{N_i, g_i} p_i$ such that $\sigma \upharpoonright \mathcal{X}_i = \chi_{N_i}$ and $\delta \models g_i$. Clearly, $\delta \models g_1 \wedge g_2$, so we are halfway there. We now want to show $\sigma = \chi_{N_1 \cup N_2}$. This is simple, because $N_1 \cap \mathcal{C} = N_2 \cap \mathcal{C}$ guarantees that functions χ_{N_1} and χ_{N_2} agree where their

domains intersect. Thus we have $\chi_{N_1 \cup N_2} = \chi_{N_1} \cup \chi_{N_2} = \sigma \upharpoonright \mathcal{X}_1 \cup \sigma \upharpoonright \mathcal{X}_2 = \sigma$.
 \square

Lemma 2. *Assume condition (b) from Definition 3 holds for two constraints ψ_{q_1} and ψ_{q_2} and automata \mathcal{A}_1 and \mathcal{A}_2 with domains \mathcal{X}_1 and \mathcal{X}_2 , respectively. Then condition (b) also holds for $\psi_{q_1} \wedge \psi_{q_2}$ and automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$.*

Proof. Given a transition $(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (p_1, p_2)$ in the product automata. Assume that we have a δ such that $\text{dom}(\delta) = \widehat{N}_1 \cup \widehat{N}_2$ and $\delta \models g_1 \wedge g_2$. Firstly, we can conclude both that $\delta \models g_i$, for $i \in \{1, 2\}$. From condition (b) of Definition 3 with respect to ψ_{q_i} , we obtain that $\chi_{N_i}, \delta^+ \cup \{\text{state}'_i \mapsto p_i\} \models \psi_{q_i}$. Now as $N_1 \cap \mathcal{C} = N_2 \cap \mathcal{C}$, we obtain $\chi_{N_1} \cup \chi_{N_2} = \chi_{N_1 \cup N_2}$, as in the proof of Lemma 1. We have immediately that $\chi_{N_1 \cup N_2}, \delta^+ \cup \{\text{state}'_1 \mapsto p_1, \text{state}'_2 \mapsto p_2\} \models \psi_{q_i}$, for $i \in \{1, 2\}$, hence $\chi_{N_1 \cup N_2}, \delta^+ \cup \{\text{state}'_1 \mapsto p_1, \text{state}'_2 \mapsto p_2\} \models \psi_{q_1} \wedge \psi_{q_2}$. \square

The Lemmas 1 and 2 show exactly our correctness result of the constraint engine with respect to the constraint automata. We make our claim precise in the following theorem.

Theorem 2. *If ψ_{q_1} encodes state q_1 from automaton \mathcal{A}_1 and ψ_{q_2} encodes state q_2 from automaton \mathcal{A}_2 , then $\psi_{q_1} \wedge \psi_{q_2}$ encodes state (q_1, q_2) from automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$.*

Proof. Follows directly from Lemmas 1 and 2. \square

5. Adding Context Dependency

We have introduced three different types of constraints to capture the behaviour of $\mathcal{R}eo$ connectors, namely synchronisation, data flow, and state constraints. We now explore one further possibility and present context dependency as an example extension that can be modelled by constraints.

One of the main contributions of the Connector Colouring (CC) framework [20] is a $\mathcal{R}eo$ semantics that expresses *context dependency*, a feature missing from the constraint automata model. A primitive depends on its context if its behaviour changes non-monotonically with increases in possibilities of data flowing on its ends. That is, by adding more possibilities of data flow, the primitive actually rules out already valid behaviour possibilities. Two important example primitives that could not be represented in previous semantic models are:

Context-dependent LossySync This channel loses data written to its source only if the surrounding context is unable to accept the data through its sink; otherwise the data flows through the channel. This corresponds to the original intention of the LossySync channel [4].

Priority merger This is a special variant of a merger that favours one of its sink ends: if data flow is possible at both sink ends, it prefers a particular end over the other.

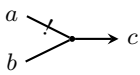
Channel	Representation	Colouring table
Context LossySync	$a \text{ --- } \text{!} \text{ --- } b$	$\begin{array}{l} a \text{ --- } \text{!} \text{ --- } b \\ a \text{ --- } \text{!} \text{ --- } b \\ a \text{ --- } \text{!} \text{ --- } b \end{array}$
Priority Merger		$\begin{array}{ll} a \text{ --- } \text{!} \text{ --- } c & a \text{ --- } \text{!} \text{ --- } c \\ b \text{ --- } \text{!} \text{ --- } c & b \text{ --- } \text{!} \text{ --- } c \\ a \text{ --- } \text{!} \text{ --- } c & a \text{ --- } \text{!} \text{ --- } c \\ b \text{ --- } \text{!} \text{ --- } c & b \text{ --- } \text{!} \text{ --- } c \end{array}$
FIFOEmpty ₁	$a \text{ --- } \square \text{ --- } b$	$\begin{array}{l} a \text{ --- } \text{!} \text{ --- } b \\ a \text{ --- } \text{!} \text{ --- } b \end{array}$

Table 2: Colouring tables for some channels

Having context dependency, as described by the CC semantics, enables a more expressive model than other \mathcal{Reo} semantics which lack this notion. Context dependency can, for example, express that data flows through a LossySync whenever the output end is able to accept the data, which could not be captured before—previous models could express only a non-deterministic choice between passing data through a LossySync or losing it. A more extensive description of the advantages of context dependency in \mathcal{Reo} can be found in previous work [20, 17]. The rest of this section gives a brief description of the connector colouring framework, and explores how to encode it in the constraint-based framework.

5.1. Connector colouring: an overview

The connector colouring (CC) semantics, as presented by Clarke et al. [20], is based on the idea of colouring the ends of a connector using a set of three colours—for orientation, the \bullet indicates the end. One colour ($\text{---}\bullet$) marks ends in the connector where data flows, and two colours mark the absence of data flow ($\text{---}\blacktriangleleft\bullet$ and $\text{---}\blacktriangleright\bullet$). The main idea is that every absence of flow must have a *reason* for excluding the flow of data, for example, that an empty FIFO₁ buffer cannot produce data. The different no-flow colours mark the direction from where the reason originates. $\text{---}\blacktriangleleft\bullet$ denotes that the reason for no-flow originates from the context, and we say that the end *requires a reason* for no flow. Similarly, $\text{---}\blacktriangleright\bullet$ indicates that the reason for no-flow originates from the primitive and we say that the end *gives a reason* for no-flow.

Colouring a connector means associating colours to each of its ends in such a way that the colours of two connected ends *match*. The colour of two ends match if both represent flow, or if the reason for no-flow comes from at least one of the ends. That is, the valid combinations are: $\text{---}\bullet\text{---}\bullet$, $\text{---}\blacktriangleleft\bullet\text{---}\blacktriangleleft\bullet$, $\text{---}\blacktriangleright\bullet\text{---}\blacktriangleright\bullet$ and $\text{---}\blacktriangleleft\bullet\text{---}\blacktriangleright\bullet$. Each primitive has only a specific set of possible *colourings*, which determine its synchronisation constraints. Each colouring is a mapping from the ends of a primitive to a colour, and the set of the colourings of a connector is called its *colouring table*. We present in Table 2 the colouring tables for some

primitives. Composition of two connectors is done by creating a new colouring table with all the possible colourings that result from matching the colours of their connected ends.

5.2. Colouring tables

Let $Colour = \{\longrightarrow, -\triangleright-\bullet, -\triangleleft-\bullet\}$ be the set of possible colours for each end, and \mathcal{X} be the global set of ends. Furthermore, for any $X \subseteq \mathcal{X}$ let $X^\updownarrow = \{x^\downarrow \mid x \in X\} \cup \{x^\uparrow \mid x \in X\}$, where x^\downarrow denotes that x is a source end, and x^\uparrow denotes that x is a sink end. We formalise colourings as follows.

Definition 4 (Colouring). [20] *A colouring $c : X^\updownarrow \rightarrow Colour$ for $X \subseteq \mathcal{X}$ is a function that maps each primitive end to a colour.*

When there the direction of a primitive end is not relevant, we write simply x instead of x^\downarrow or x^\uparrow . A colouring identifies a step of the execution of $\mathcal{R}eo$ connector, disregarding any data constraint. For example, the colouring $c_1 = \{a^\downarrow \mapsto \longrightarrow, b^\uparrow \mapsto -\triangleright-\bullet\}$, which we also write as ' $a \bullet \longrightarrow \triangleright \bullet b$ ', describes a scenario where the channel end a has dataflow and the channel end b does not have data flow. Furthermore, b provides a reason for the absence of dataflow. We drop the superscripts \uparrow and \downarrow on the port names because these can be inferred by matching the image with the graphical representation of the primitive. A collection of colourings yields a colouring table, that describes a the possible behaviour of a connector.

Definition 5 (Colouring Table). [20] *A colouring table over ports $X \subseteq \mathcal{X}$ is a set of colourings with domain X^\updownarrow .*

The colouring c_1 described above represents one of the possible behaviours of the empty $FIFO_1$ channel, with source port a and sink port b . Another possible colouring of this channel is $c_2 = a \bullet \triangleright - \triangleright \bullet b$. For the empty $FIFO_1$ channel, it's colouring table consists of $\{c_1, c_2\}$, describing all the possible steps that this channel can perform, as defined in Table 2.

Recall that we colour a $\mathcal{R}eo$ connector by selecting colourings of each primitive in the connector and verifying that the colours of shared ends match. We formalise this process by defining the product of colouring tables, presented below. We use the symbol ' \circ ' to range over the set $\{\uparrow, \downarrow\}$, and define $\bar{\uparrow} = \downarrow$ and $\bar{\downarrow} = \uparrow$.

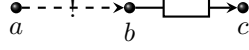
Definition 6 (Product). *The product of two colouring tables T_1 and T_2 , denoted by $T_1 \odot T_2$, is the following colouring table:*

$$\{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, \\ x^\circ \in dom(c_1) \wedge x^{\bar{\circ}} \in dom(c_2) \Rightarrow c_1(x^\circ) \cap c_2(x^{\bar{\circ}})\}.$$

The binary relation \cap consists only of valid matches of colourings:

$$\cap = \{\langle \longrightarrow, \longrightarrow \rangle, \langle -\triangleleft-\bullet, -\triangleright-\bullet \rangle, \langle -\triangleright-\bullet, -\triangleleft-\bullet \rangle, \langle -\triangleright-\bullet, -\triangleright-\bullet \rangle\}$$

Example 7. We present as a simple example the connector resulting from the composition of a context-dependent *LossySync* with a *FIFOEmpty₁* channel:



Composing the colouring tables of both primitives, presented in Table 2, results in the colouring table of the connector, illustrated as follows:



Each image represents a valid colouring, and only these two colourings exist for this connector. The first colouring corresponds to the flow of data through the context-dependent *LossySync* and into the *FIFO₁* buffer, and the second colouring corresponds to the absence of flow in the connector, with a reason for this absence required from end *a*. In both colourings, end *c* of the *FIFO₁* buffer gives a reason for no data flow.

Example 7 illustrates that data flowing into the context-dependent *LossySync* can not be lost if there is a primitive or component willing to accept that data at its sink end. Furthermore, it also illustrates that an empty *FIFO₁* buffer can never produce data, and therefore always gives a reason for no flow to the context.

5.3. Context constraints

To capture context dependency, the constraint-based semantics are extended with an extra set of *context constraints* defined in terms of synchronisation variables (\mathcal{X}) and a new set of variables called *context variables*. The flow axiom is also updated to link the two sets of variables.

Context variables represent the direction of the reasons for no-flow, and context constraints reflect the valid combinations for the context variables. Context variables are given by the following set:

$$\{x_{\text{sk}} \mid x \in \mathcal{X}\} \cup \{x_{\text{sr}} \mid x \in \mathcal{X}\}$$

Use x_{sk} when the end x is a sink end, and x_{sr} when the end x is a source end. We also write X_{sk} and X_{sr} to denote the sets $\{x_{\text{sk}} \mid x \in X\}$ and $\{x_{\text{sr}} \mid x \in X\}$, respectively. Note that the constraints on variables x_{sk} and x_{sr} typically occur in different primitives—the two primitives connected at node x . Thus for each end in a closed connector, we have constraints defined in terms of variables x , \hat{x} , x_{sk} and x_{sr} . The intention is that x_{sk} or x_{sr} is true when the end x gives a reason and false if x requires a reason. The values of these variables are unimportant when there is flow on x .

Next, we extend the flow axiom to reflect the matching of reasons:

$$(\neg x \leftrightarrow \hat{x} = \text{NO-FLOW}) \wedge (\neg x \rightarrow x_{\text{sk}} \vee x_{\text{sr}}) \quad (\text{updated flow axiom})$$

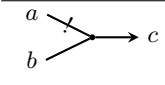
Channel	Context Constraints
$a \dashrightarrow b$	$\neg a \rightarrow (\neg b \wedge \neg a_{sr} \wedge b_{sk}) \wedge \neg b \rightarrow ((a \wedge \neg b_{sk}) \vee \neg a)$
	$(c \wedge \neg a) \rightarrow \neg a_{sr} \wedge$ $(c \wedge \neg b) \rightarrow b_{sr} \wedge \neg c \rightarrow ((\neg a_{sr} \wedge \neg b_{sr}) \vee \neg c_{sk})$
$a \boxed{\rightarrow} b$	$(\neg a \rightarrow \neg a_{sr}) \wedge b_{sk}$

Table 3: Context constraints for the channels presented in Table 2.

Recall that channels are composed using the same name x for a source and a sink end. The constraint $x_{sk} \vee x_{sr}$ can be interpreted as follows: The reason for no data flow can either come from the sink end (x_{sk} is true), come from the source end (x_{sr} is true), or from both ends at the same time, but the reason can never come from nowhere (both x_{sk} and x_{sr} are false). The constraint thus encodes the three valid matching no-flow possibilities given above.

The context constraints for the primitives shown in Section 5.1 are presented in Table 3. The other primitives in Table 1 must also be extended to reflect the valid combinations of context variables, by encoding their colouring tables (see Clarke *et al.* [20] for examples).

Example 8. Recall Example 7, where a context-dependent *LossySync* is composed with *FIFOEmpty₁* channel. The constraints of this connector are as follows:

$$\begin{aligned}
\Psi_{SC} &= b \rightarrow a \wedge \neg c \\
\Psi_{DFC} &= b \rightarrow (\hat{a} = \hat{b}) \wedge \top \\
\Psi_{CC} &= \neg a \rightarrow (\neg b \wedge \neg a_{sr} \wedge b_{sk}) \wedge \neg b \rightarrow ((a \wedge \neg b_{sk}) \vee \neg a) \wedge (\neg b \rightarrow \neg b_{sr}) \wedge c_{sk} \\
\Psi &= \exists b, \hat{b}, a_{sr}, a_{sk}, b_{sr}, b_{sk}, c_{sr}, c_{sk}. (\Psi_{SC} \wedge \Psi_{DFC} \wedge \Psi_{CC} \wedge Flow(\{a, b, c\}))
\end{aligned}$$

The variables $\{a_{sr}, a_{sk}, b_{sr}, b_{sk}, c_{sr}, c_{sk}\}$ are also hidden by the existential quantifier as they are necessary only to exclude non-solutions, such as the context-dependent *LossySync* losing data without any reason, or the *FIFOEmpty₁* failing to receive a value without a reason. The actual values chosen for such variables do not matter.

A SAT solver can solve the constraint Ψ_{SC} yielding the solutions:

$$\sigma_1 = a \wedge b \wedge \neg c \quad \sigma_2 = a \wedge \neg b \wedge \neg c \quad \sigma_3 = \neg a \wedge \neg b \wedge \neg c$$

Using these solutions, we can simplify Ψ using standard techniques, as in Section 3.4, to derive the corresponding data flow constraints. The novelty in this setting is the context constraints. Using the solutions σ_1, σ_2 and σ_3 to simplify Ψ , we obtain the following constraints on the context variables (which are hidden by the existential quantifier):

$$\begin{aligned}
\Psi \wedge \sigma_1 &\rightsquigarrow c_{sk} \\
\Psi \wedge \sigma_2 &\rightsquigarrow \perp \\
\Psi \wedge \sigma_3 &\rightsquigarrow a_{sk} \wedge \neg a_{sr} \wedge b_{sk} \wedge \neg b_{sr} \wedge c_{sk}
\end{aligned}$$

σ	$f(\sigma)$	σ	$f(\sigma)$
x	$x^\downarrow \mapsto \longrightarrow \bullet$	x	$x^\uparrow \mapsto \longrightarrow \bullet$
$\neg x \wedge x_{\text{sr}}$	$x^\downarrow \mapsto -\blacktriangleright-\bullet$	$\neg x \wedge x_{\text{sk}}$	$x^\uparrow \mapsto -\blacktriangleright-\bullet$
$\neg x \wedge \neg x_{\text{sr}}$	$x^\downarrow \mapsto -\blacktriangleleft-\bullet$	$\neg x \wedge \neg x_{\text{sk}}$	$x^\uparrow \mapsto -\blacktriangleleft-\bullet$

Table 4: Definition of f .

Only the first and the last constraints are satisfiable. We can conclude that:

- σ_1 is a solution that gives a reason on the end c , without imposing any restrictions on the value c_{sr} , as $c_{\text{sk}} \rightarrow c_{\text{sk}} \vee c_{\text{sr}}$;
- σ_2 is not a valid solution, i.e., it is not possible for the context-dependent LossySync to lose data since the FIFOEmpty₁ does not provide a reason for losing data; and
- σ_3 is a valid solution where no data flow occurs in the channels, provided that there is a reason given to a , i.e., that a_{sk} is true. As with σ_1 , no restrictions are imposed on the value of c_{sr} .

5.4. Correctness of context constraints

This approach to context dependency is equivalent to the 3-colouring semantics of Clarke *et al.* [20]. It does not try to overcome any of the limitations of the 3-colouring, such as problems related to causality, although it does permit the description of data constraints, which is not present in the CC framework. The equivalence of our approach with the 3-colouring follows by construction. The proof of this equivalence results from the following observations:

1. the constraints of each primitive p are defined so that there is a surjection f , defined in Table 4, from the solutions of the synchronisation and context constraints onto the entries of the colouring table of p ; and
2. f is compositional, namely, when composing two primitives p and q with a shared variable x , composing their colouring tables and applying f to find the possible solutions is equivalent to apply f to each colouring table and then finding the solutions for the conjunction of these constraints and the updated flow axiom.

Let σ be an assignment. For each pair $x, x_s \in \text{dom}(\sigma)$, where $s \in \{\text{sr}, \text{sk}\}$, $f(\sigma)$ is the colouring table that maps x^\downarrow or x^\uparrow to the colouring represented in Table 4. Furthermore, by the definition of f every colouring is associated to a fixed set of possible solutions, which can be trivially written as a constraint. That is, the inverse function f^{-1} will always produce solutions of possible constraints. It can be easily shown that the colouring tables of the primitives in Table 2 are obtained by applying f to the solutions of the synchronisation and context constraints presented in Table 3. For example, the synchronisation and

context constraints for the FIFOEmpty_1 are $\neg b \wedge (\neg a \rightarrow \neg a_{\text{sr}}) \wedge b_{\text{sk}}$, and its possible solutions are:

$$\begin{aligned}\sigma_1 &= a \wedge a_{\text{sr}} \wedge \neg b \wedge b_{\text{sk}}; \\ \sigma_2 &= a \wedge \neg a_{\text{sr}} \wedge \neg b \wedge b_{\text{sk}}; \text{ and} \\ \sigma_3 &= \neg a \wedge \neg a_{\text{sr}} \wedge \neg b \wedge b_{\text{sk}}.\end{aligned}$$

It follows from the definition of f that $f(\sigma_1) = f(\sigma_2) = a \longleftarrow \triangleright \rightarrow b$, and $f(\sigma_3) = a \longleftarrow \triangleright - \triangleright \rightarrow b$, which is what we expect.

Observe that for every $\mathcal{R}\text{eo}$ connector the set of variables used in its synchronisation and context constraints must obey certain properties, captured by the notion of *variable set* defined below.

Definition 7. A variable set is a set $V \subseteq \mathcal{X} \cup \mathcal{X}_{\text{sr}} \cup \mathcal{X}_{\text{sk}}$ such that

$$x \in V \cap \mathcal{X} \quad \text{iff} \quad x_{\text{sr}} \in V \cap \mathcal{X}_{\text{sr}} \vee x_{\text{sk}} \in V \cap \mathcal{X}_{\text{sk}}.$$

We say two sets of variables V_1 and V_2 are *compatible*, written as $V_1 \frown V_2$, if V_1 and V_2 are variable sets, and for every $x \in V_1 \cap V_2 \cap \mathcal{X}$, x is a sink end in V_1 and a source end in V_2 or vice-versa, that is,

$$V_1 \frown V_2 \quad \text{iff} \quad V_1 \cap V_2 \subseteq \mathcal{X} \tag{2}$$

Note that if $V_1 \frown V_2$ then $V_1 \cup V_2$ is also a variable set. Intuitively, two connectors with constraints over the variable sets V_1 and V_2 can only be composed if the source and sink ends are connected in a 1:1 fashion. Recall that the composition of two $\mathcal{R}\text{eo}$ connectors, introduced in Section 3.4, requires that every shared end of the composed connectors is a source end in one of the connectors, and a sink end in the other connector.

We now assume that, for every primitive p , the colouring table is given by the surjection f defined in Table 4 with respect to the solutions of the synchronisation and context constraints. Let $fv(\cdot)$ be a function that returns the free variables of a constraint. A constraint Ψ is defined over a variable set V whenever $fv(\Psi) \subseteq V$. Let also $\llbracket \cdot \rrbracket_V$ be a function that yields the set of all possible solutions of Ψ over V , that is,

$$\llbracket \Psi \rrbracket_V = \{ \sigma \mid \sigma \models \Psi, \text{dom}(\sigma) = V \}.$$

Define F to be the lifting of f to sets, i.e., $F(\Sigma) = \{ f(\sigma) \mid \sigma \in \Sigma \}$. Finally, let \boxtimes denote the composition of two synchronisation and context constraints Ψ_1 over the set V_1 and Ψ_2 over the set V_2 , where $V_1 \frown V_2$, defined as follows:

$$\Psi_1 \boxtimes \Psi_2 = \Psi_1 \wedge \Psi_2 \wedge \bigwedge_{x \in V_1 \cap V_2} (\neg x \rightarrow x_{\text{sk}} \vee x_{\text{sr}}),$$

where the last constraint reflects the update on the flow axiom. The correctness of the composition of our encoding as constraints with respect to the connector colouring semantics is formalised by the following theorem.

Theorem 3. For any pair of constraints Ψ_p over the variable set V_1 and Ψ_q over the variable set V_2 such that $V_1 \frown V_2$, the following holds:

$$F(\llbracket \Psi_p \rrbracket_{V_1}) \odot F(\llbracket \Psi_q \rrbracket_{V_2}) = F(\llbracket \Psi_p \sqcap \Psi_q \rrbracket_{V_1 \cup V_2}).$$

Before proving this theorem we prove four helper lemmas. The first lemma relates shared variables of $\mathcal{R}\text{eo}$ connectors and the domain of the colourings derived from specific solutions of the same connectors. In the following we use the symbol \circ to range over $\{\uparrow, \downarrow\}$, and define $\bar{\uparrow} = \downarrow$ and $\bar{\downarrow} = \uparrow$.

Lemma 3. Let σ_1 and σ_2 be assignments such that $\text{dom}(\sigma_i) = V_i$ for $i \in 1..2$, and $V_1 \frown V_2$. Then the following holds.

$$x^\circ \in \text{dom}(f(\sigma_1)) \wedge x^{\bar{\circ}} \in \text{dom}(f(\sigma_2)) \quad \text{iff} \quad x \in V_1 \cap V_2.$$

Proof. Let x be such that $x^\circ \in \text{dom}(f(\sigma_1))$ and $x^{\bar{\circ}} \in \text{dom}(f(\sigma_2))$. By the definition of f , if $x^\circ \in \text{dom}(f(\sigma_1))$ then x must occur also in $\text{dom}(\sigma_1)$, and similarly x must occur also in $\text{dom}(\sigma_2)$. Therefore $x \in V_1 \cap V_2$. For the other implication assume that $x \in V_1 \cap V_2$. Also $x \in \mathcal{X}$ because $V_1 \frown V_2$. By the definitions of f and because $V_1 \frown V_2$ (and therefore V_1 and V_2 are variable sets) we conclude that $x^\uparrow \in \text{dom}(f(\sigma_1))$ and $x^\downarrow \in \text{dom}(f(\sigma_2))$ or $x^\downarrow \in \text{dom}(f(\sigma_1))$ and $x^\uparrow \in \text{dom}(f(\sigma_2))$. \square

We say two assignments σ_1 and σ_2 are compatible, written as $\sigma_1 \frown \sigma_2$, iff $\forall x \in \text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) \cdot \sigma_1(x) = \sigma_2(x)$. The second lemma shows a sufficient condition for any two assignments σ_1 and σ_2 be compatible.

Lemma 4. Let σ_1 and σ_2 be assignments where $\text{dom}(\sigma_1) = V_1$, $\text{dom}(\sigma_2) = V_2$, $V_1 \frown V_2$, and $\forall x \in V_1 \cap V_2 \cdot f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^{\bar{\circ}})$. Then

$$\sigma_1 \frown \sigma_2.$$

As a corollary, we have that $f(\sigma_1) \cup f(\sigma_2) = f(\sigma_1 \cup \sigma_2)$.

Proof. We prove $\sigma_1 \frown \sigma_2$ by contrapositive, showing that if $\neg(\sigma_1 \frown \sigma_2)$ then $\exists x \in V_1 \cap V_2 \cdot \neg(f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^{\bar{\circ}}))$. Assuming $\neg(\sigma_1 \frown \sigma_2)$, there exists $x \in V_1 \cap V_2$ such that $\sigma_1(x) \neq \sigma_2(x)$. Note that from Equation 2 we conclude that $x \in \mathcal{X}$. Without loss of generality assume $\sigma_1(x) = \top$ and $\sigma_2(x) = \perp$. Then $f(\sigma_1)(x^\circ) = \longrightarrow$ and $f(\sigma_2)(x^{\bar{\circ}}) \neq \longrightarrow$, for some $\circ \in \{\uparrow, \downarrow\}$. Thus $\neg(f(\sigma_1) \frown f(\sigma_2))$. Similarly for $\sigma_1(x) = \perp$, $\sigma_2(x) = \top$. By contrapositive we conclude that $\sigma_1 \frown \sigma_2$. \square

The next lemma relates the new constraint from the updated flow axiom to the matching of colourings.

Lemma 5. The constraint $\neg x \rightarrow (x_{\text{sk}} \vee x_{\text{sr}})$ and the matching of colours are related by the condition below, where $\{x, x_{\text{sr}}, x_{\text{sk}}\} \subseteq V$:

$$c \in F(\llbracket \neg x \rightarrow (x_{\text{sr}} \vee x_{\text{sk}}) \rrbracket_V) \quad \text{iff} \quad c(x^\downarrow) \frown c(x^\uparrow).$$

Proof. The proof follows by unfolding the definitions of F , and applying Lemma 4.

$$\begin{aligned}
& c \in F(\llbracket \neg x \rightarrow (x_{\text{sr}} \vee x_{\text{sk}}) \rrbracket_V) \\
& \langle \text{By the definition of } F \rangle \\
\equiv & c = f(\sigma), \sigma \models \neg x \rightarrow (x_{\text{sr}} \vee x_{\text{sk}}), \text{ and } \text{dom}(\sigma) = V \\
& \langle \text{Partition } \sigma \text{ into } \sigma' \text{ and } \sigma'' \text{ such that } \text{dom}(\sigma') = \{x, x_{\text{sr}}, x_{\text{sk}}\} \rangle \\
\equiv & c = f(\sigma' \cup \sigma''), \sigma' \models \neg x \rightarrow (x_{\text{sr}} \vee x_{\text{sk}}), \text{ and } \text{dom}(\sigma'') = V \setminus \{x, x_{\text{sr}}, x_{\text{sk}}\}
\end{aligned}$$

Observe now that $\sigma' \frown \sigma''$, hence we know that $f(\sigma' \cup \sigma'') = f(\sigma') \cup f(\sigma'')$. Furthermore, the possible solutions for σ' are the following:

$$\begin{array}{cccc}
x \wedge x_{\text{sr}} \wedge x_{\text{sk}} & x \wedge \neg x_{\text{sr}} \wedge x_{\text{sk}} & \neg x \wedge x_{\text{sr}} \wedge x_{\text{sk}} & \neg x \wedge \neg x_{\text{sr}} \wedge x_{\text{sk}} \\
x \wedge x_{\text{sr}} \wedge \neg x_{\text{sk}} & x \wedge \neg x_{\text{sr}} \wedge \neg x_{\text{sk}} & \neg x \wedge x_{\text{sr}} \wedge \neg x_{\text{sk}} & \neg x \wedge \neg x_{\text{sr}} \wedge \neg x_{\text{sk}}
\end{array}$$

Let Σ be this set. The last step of the proof above is equivalent to

$$c = f(\sigma') \cup f(\sigma''), \sigma' \in \Sigma, \text{ and } \text{dom}(\sigma'') = V \setminus \{x, x_{\text{sr}}, x_{\text{sk}}\}.$$

Observe that $f(\sigma')$, for $\sigma' \in \Sigma$, are exactly the set of all possible colourings c' such that $c'(x^\downarrow) \frown c'(x^\uparrow)$, where $\text{dom}(c') = \{x^\downarrow, x^\uparrow\}$. It is now enough to observe that $f(\sigma'')$ yields any possible colouring, assigning colours to all the remaining ends apart from x , not mentioned on the right-hand-side of the equivalence we want to prove. \square

Our final lemma relates the matching of the colouring yield by two different assignments and a new assignment that satisfies the constraint $\neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}$.

Lemma 6. For σ_1 and σ_2 such that $\text{dom}(\sigma_i) = V_i$, for $i \in \{1, 2\}$, $\sigma_1 \frown \sigma_2$, and $V_1 \frown V_2$:

$$\begin{aligned}
& f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^\circ) \text{ iff} \\
& (\sigma_1 \cup \sigma_2) \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}} \text{ and } \{x, x_{\text{sr}}, x_{\text{sk}}\} \subseteq \text{dom}(\sigma_1 \cup \sigma_2).
\end{aligned}$$

Proof. For simplicity let $c_1 = f(\sigma_1)$ and $c_2 = f(\sigma_2)$. Let also $\{x, x_{\text{sr}}, x_{\text{sk}}\} \subseteq V$. Note that c_1 and c_2 will have disjoint domains, because $V_1 \frown V_2$. Observe that:

$$\begin{aligned}
& c_1(x^\circ) \frown c_2(x^\circ) \\
& \langle \text{Because } \text{dom}(c_1) \text{ and } \text{dom}(c_2) \text{ are disjoint} \rangle \\
= & (c_1 \cup c_2)(x^\circ) \frown (c_1 \cup c_2)(x^\circ) \\
& \langle \text{By Lemma 5} \rangle \\
= & c_1 \cup c_2 \in F(\llbracket \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}} \rrbracket_V) \\
& \langle \text{By the definition of } F \text{ and } \llbracket \cdot \rrbracket_V \rangle \\
= & c_1 \cup c_2 \in \{f(\sigma) \mid \sigma \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}, \text{dom}(\sigma) = V\} \\
& \langle \text{Because } \sigma_1 \frown \sigma_2 \text{ and } c_i = f(\sigma_i) \rangle \\
= & f(\sigma_1 \cup \sigma_2) \in \{f(\sigma) \mid \sigma \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}, \text{dom}(\sigma) = V\} \\
& \langle \text{By set inclusion and because } V \subseteq \{x, x_{\text{sr}}, x_{\text{sk}}\} \rangle \\
= & (\sigma_1 \cup \sigma_2) \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}} \text{ and } \{x, x_{\text{sr}}, x_{\text{sk}}\} \subseteq \text{dom}(\sigma_1 \cup \sigma_2)
\end{aligned}$$

□

Proof. (Theorem 3)

$$\begin{aligned}
& F(\llbracket \Psi_p \rrbracket)_{V_1} \odot F(\llbracket \Psi_q \rrbracket)_{V_2} \\
& \quad \langle \text{By the definition of } F \rangle \\
& = \{f(\sigma_1) \mid \sigma_1 \in \llbracket \Psi_p \rrbracket_{V_1}\} \odot \{f(\sigma_2) \mid \sigma_2 \in \llbracket \Psi_q \rrbracket_{V_2}\} \\
& \quad \langle \text{By the definition of } \llbracket \cdot \rrbracket_V \rangle \\
& = \{f(\sigma_1) \mid \sigma_1 \models \Psi_p, \text{dom}(\sigma_1) = V_1\} \odot \{f(\sigma_2) \mid \sigma_2 \models \Psi_q, \text{dom}(\sigma_2) = V_2\} \\
& \quad \langle \text{By the definition of } \odot \rangle \\
& = \{f(\sigma_1) \cup f(\sigma_2) \mid \sigma_1 \models \Psi_p, \sigma_2 \models \Psi_q, \text{dom}(\sigma_1) = V_1, \text{dom}(\sigma_2) = V_2, \\
& \quad x^\circ \in \text{dom}(f(\sigma_1)) \wedge x^\circ \in \text{dom}(f(\sigma_2)) \Rightarrow f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^\circ)\} \\
& \quad \langle \text{By Lemma 3} \rangle \\
& = \{f(\sigma_1) \cup f(\sigma_2) \mid \sigma_1 \models \Psi_p, \sigma_2 \models \Psi_q, \text{dom}(\sigma_1) = V_1, \text{dom}(\sigma_2) = V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^\circ)\} \\
& \quad \langle \text{By Lemma 4, from where we also conclude that } \sigma_1 \frown \sigma_2 \rangle \\
& = \{f(\sigma_1 \cup \sigma_2) \mid \sigma_1 \models \Psi_p, \sigma_2 \models \Psi_q, \text{dom}(\sigma_1 \cup \sigma_2) = V_1 \cup V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^\circ)\} \\
& \quad \langle \text{By Lemma 6, and because } \sigma_1 \frown \sigma_2 \text{ and } \text{dom}(\sigma_1 \cup \sigma_2) = V_1 \cup V_2 \rangle \\
& = \{f(\sigma_1 \cup \sigma_2) \mid \sigma_1 \models \Psi_p, \sigma_2 \models \Psi_q, \text{dom}(\sigma_1 \cup \sigma_2) = V_1 \cup V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow (\sigma_1 \cup \sigma_2) \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}\} \\
& \quad \langle \text{Because } \sigma_1 \frown \sigma_2 \rangle \\
& = \{f(\sigma_1 \cup \sigma_2) \mid (\sigma_1 \cup \sigma_2) \models \Psi_p, (\sigma_1 \cup \sigma_2) \models \Psi_q, \text{dom}(\sigma_1 \cup \sigma_2) = V_1 \cup V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow (\sigma_1 \cup \sigma_2) \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}\} \\
& \quad \langle \text{Using } \sigma = \sigma_1 \cup \sigma_2 \text{ and replacing implication by an universal quantifier} \rangle \\
& = \{f(\sigma) \mid \sigma \models \Psi_p, \sigma \models \Psi_q, \text{dom}(\sigma) = V_1 \cup V_2, \forall x \in V_1 \cap V_2 \cdot \sigma \models \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}\} \\
& \quad \langle \text{Because } \sigma \models \Psi_1 \text{ and } \sigma \models \Psi_2 \text{ iff } \sigma \models \Psi_1 \wedge \Psi_2 \rangle \\
& = \{f(\sigma) \mid \sigma \models \Psi_p \wedge \Psi_q \wedge \bigwedge_{x \in V_1 \cap V_2} \neg x \rightarrow x_{\text{sr}} \vee x_{\text{sk}}, \text{dom}(\sigma) = V_1 \cup V_2\} \\
& \quad \langle \text{By the definition of } \Box \rangle \\
& = \{f(\sigma) \mid \sigma \models \Psi_p \Box \Psi_q, \text{dom}(\sigma) = V_1 \cup V_2\} \\
& \quad \langle \text{By the definition of } F \text{ and } \llbracket \cdot \rrbracket_V \rangle \\
& = F(\llbracket \Psi_p \Box \Psi_q \rrbracket)_{V_1 \cup V_2}.
\end{aligned}$$

□

6. Benchmarks

We compare two prototype engines based on constraint satisfaction with an optimised engine for \mathcal{Reo} based on the Connector Colouring semantics [20]. In the three cases the data constraints are ignored, and solving the constraints

yields only *where* data can flow, but not *which* data flows. We evaluate the constraints-based approach using implementations based on both context independent (CI) and context dependent (CD) semantics (Section 5). In the connector colouring semantics, the CI semantics corresponds to using two colours while the CD semantics corresponds to using three colours. We have implemented the context dependent semantics for only one of the constraint engines, as the results we present already provide solid evidence that the constraint-based approach is significantly better than using Connector Colouring. A few more words regarding the three engines follow:

CC engine We use an optimised engine based on Connector Colouring [20] as a reference for our benchmark results. This engine has already been incorporated into the Eclipse Coordination Tools.⁹ This engine supports both context dependent and independent semantics, as explained in Section 5, and it is, to the best of our knowledge, the fastest pre-existing implementation that computes the behaviour of $\mathcal{R}eo$ connectors on-the-fly.

SAT engine This is a constraint engine using the SAT4J Java libraries,¹⁰ which are free and included in the Eclipse standard libraries. The main concern of the project responsible for the SAT4J libraries is the efficiency of the SAT solver. We chose SAT4J because it is a well known library for SAT solving, with the portability advantages provided by the Java platform. We avoid solutions where no data flows by adding conjunctively the constraint $\bigvee_{x \in \mathcal{X}} x$ to the constraints of the connector for the CI semantics. We do not add this imposition in the CD semantics because the context dependency already guarantees that the data flow unless there is an explicit reason that forbids the flow of data.

CHOCO engine We developed a second prototype constraint engine using the CHOCO open-source constraint solver¹¹ which offers, among other things, good support for the end user, state-of-the-art algorithms and techniques, and user-defined constraints, domains and variables. In the future, we expect to achieve finer control over the strategies for solving the constraints, and to add the support for non-boolean variables. The CHOCO-based engine implements only the context independent (CI) semantics. We avoid solutions with no flow by using a strategy that gives precedence to solutions where the synchronisation variables are set to *true*. Using this strategy, the solution with no flow at all in the connector can still be found, but only when it is the only valid solution.

6.1. Test cases

We present four test cases constructed out of stateless channels. For each test case we replicate part of the connector n times and measure the time taken to

⁹<http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools>

¹⁰<http://www.sat4j.org/>

¹¹<http://choco.emn.fr/>

find a solution for the coordination problem, which includes the time to required build the data structures corresponding to the connector and the encoding of its behaviour as constraints. We also add *active environments* to the connector, i.e., we attach data writers and data readers to every port where we expect data to be written or read, respectively. Defining this environment is important for the CC engine because it reduces the number of possible solutions, and because one of the optimisations of the CC engine is to start computing the colouring table starting from the sources of data. Incorporating an active environment significantly reduces the time taken by the CC engine, allowing for a fairer comparison.

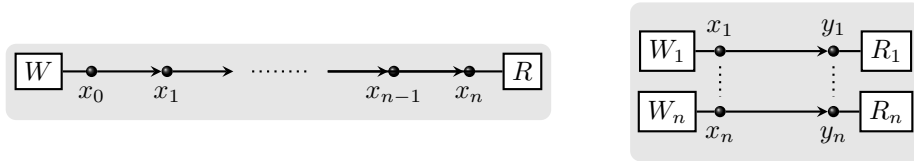


Figure 5: n synchronous channels composed in sequence (left) and in parallel (right).

The first two test cases consist of n synchronous channels in sequence (SEQ) and in parallel (PAR), respectively, as depicted in the left and right of Figure 5. Note that a sequence of n synchronous channels is semantically equivalent to a single synchronous channel, but the search for a solution becomes more complex, especially when the topology of the connector is not exploited.

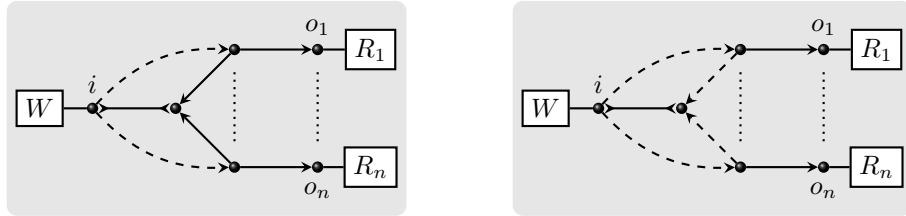


Figure 6: Exclusive (left) and inclusive (right) router connectors generalised for n outputs.

The next test case is a generalisation of the exclusive router (ExR), introduced in Section 5, for n outputs. The generalised exclusive router is depicted on the left side of Figure 6. It passes data from the data writer to exactly one of the data readers. Finally, we use a variation of the exclusive router, the *inclusive router* (InR), depicted in the right side of Figure 6. The inclusive router replicates data provided by the writer to at least one of the data readers. The number of possible solutions increases exponential when using the CI semantics, which favours the constraint engine (since only one solution is computed). When using the CD semantics with the active environment there is only one solution for the constraints, which consists of the data being replicated to all available readers.

6.2. Results

All the benchmarks were executed on a Macbook laptop with a 2 GHz Intel Core 2 Duo processor and 4 GB of RAM, running Mac OS 10.6. For each value, we performed 10 different executions and used the average value. We test the CC-, SAT-, and CHOCO-based engines using the CI semantics, and the CC- and SAT-based engines using the CD semantics. The benchmark results for the CC engine are presented in a single graph in Figure 7, and the benchmark results for the constraint engines are presented in Figure 8, using one graph for each test case. In the graphs we write **Seq** for the sequence of Sync channels, **Par** for the set of Sync channels in parallel, **ExR** for the exclusive router, and **InR** for the inclusive router. For each engine, semantics (CI and CD), and connector, we selected a range of possible sizes for the connector. For each size, we measured the time for finding a solution 10 times, and represented the average value in the graph using a marking, as explained in the legend of the graphs. Furthermore, a solid line represents the evaluation of the executions using the context independent semantics (CI), and a dashed line represents the evaluation of executions using the context dependent semantics (CD).

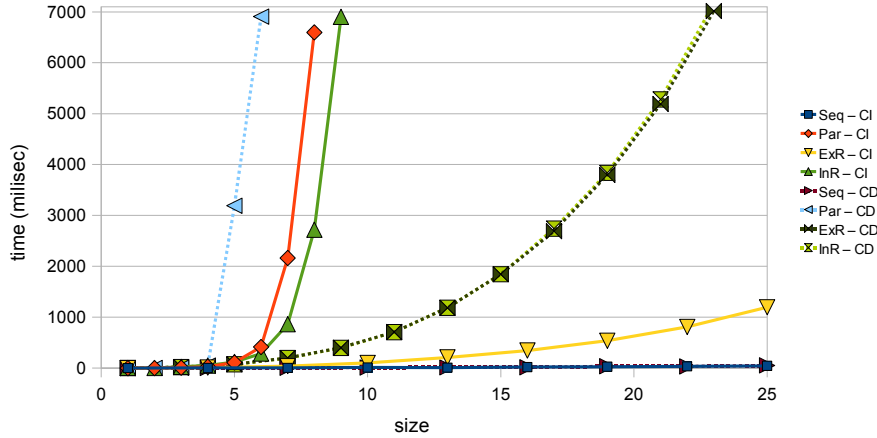


Figure 7: Result from the execution of the CC engine for each of the 4 case studies, and with both the context dependent and independent semantics.

The results show that implementations based on constraint-solving techniques are more scalable and more efficient than the implementation based on connector colouring.

Firstly, the maximum connector size for the constraint solving approach is much bigger than for the CC-based implementation. We measured the biggest connectors until running out of memory or taking more than 1 minute. While the maximum size of the connectors tested by the CC engine range between 6 (Par-CD) and 100 (Par-CI), for the CHOCO engine these values ranged between 2,000 (InR-CI) and 17,000 (Par-CI), and for the SAT engine these values range between 800 (InR-CD) and 80,000 (Par-CI).

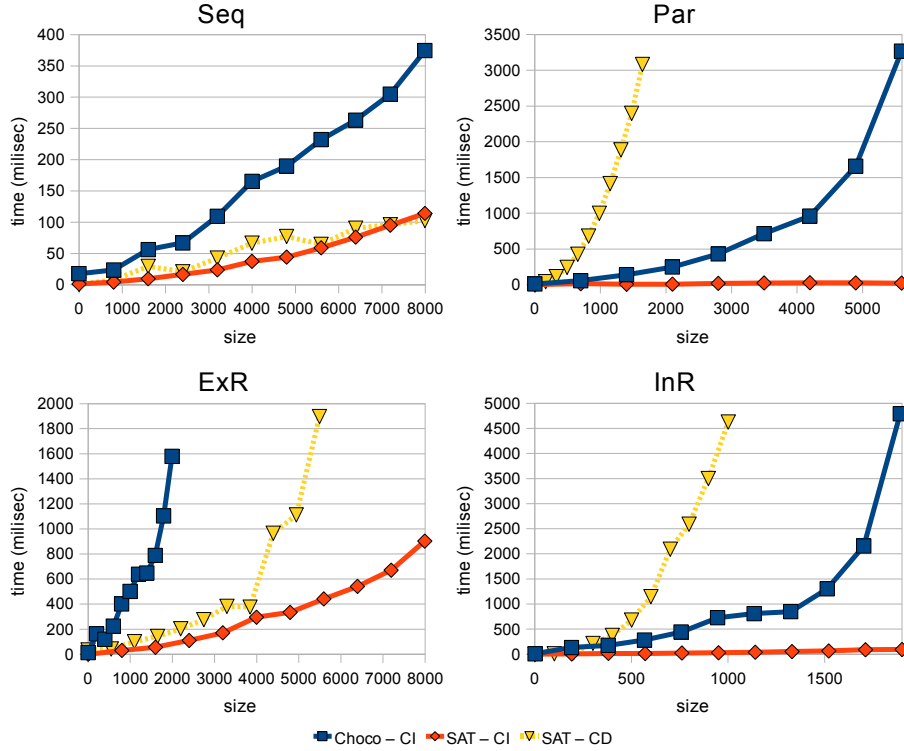


Figure 8: Result from executing the CHOCO and SAT engines for the 4 case studies.

Secondly, the constraint solver-based engines are significantly faster at finding solutions, as can easily be seen from the graphs. We present a comparison based on the *workload*—size of connector—an engine can handle within one second. For the Seq benchmark, the CC-based implementations could only handle connectors of size 89 (CC-CD) to 94 (CC-CI) in 1 second, in contrast to connectors of size 15,500 (CHOCO-CI), 28,000 (SAT-CI) and 27,500 (SAT-CD) in the constraint-based implementations. This means that the increase in workload ranges from 164 to 308. The case for the Par benchmark is more impressive: 5 (CC-CD), 7 (CC-CI) vs. 41300 (CHOCO-CI), 55000 (SAT-CI), and 1000 (SAT-CD). Thus, the increase in workload ranges from 200 to 7857. Notice that as the context dependent semantics require more variables to encode, they take significantly longer to solve in the constraint-based approach compared to context independent semantics. The ExR and InR benchmarks exhibit similar increases in workload, being able to deal with connectors that are between 46 and 1000 times larger in one second.

The main difference between execution of the CC engine and of the constraint engines is that the former takes into account the topology of the connector, and

calculates all possible solutions whenever a new primitive is added to the set of constraints. The constraint engines disregard the topology of the connector, and return only one possible solution. As a consequence, the CC engine favours connectors with a smaller number of solutions, while the constraint engine favours connectors where more solutions can be found. The inclusive router test case illustrates this point very clearly. For the CC engine the use of a context dependent semantics reduces the number of possible solutions, and for that reason the engine is much more efficient than using a context independent semantics. For the SAT engine the number of solutions has the opposite effect, since it is much faster to find a solution under the context independent semantics. This test case also shows that, even though the context dependency requires more complex reasoning, the number of solutions is more relevant for the efficiency of the engine.

Consider now Figure 8. The difference between results obtained by the CHOCO-based engine and the SAT-based engine are smaller in the connector with a sequence of Sync channels. This is due to the way we avoid no-flow solutions in both cases. By avoiding no-flow solutions, there is only one possible solution to the constraints consisting of data flowing from the writer to the reader. In the CHOCO engine we give precedence to the flow on the ends, so the engine only has to verify that it is in fact a solution. The SAT engine starts by trying the no-flow assignment for each end, with the additional constraint that at least one end has to have flow, which is not optimal for this scenario. And the context has little influence in this case because the variables that deal with the context are not relevant (and thus unconstrained) when the synchronisation variables are set to true.

When performing the benchmarking we also realised that in certain cases the solution is found unusually rapidly. This happened quite frequently, for example, when executing the inclusive router using the context dependent semantics. The existence of these *lucky* runs reflects that the heuristics used by the constraint solver are not as predictable as the compositional method used by the CC engine. The graphs do not exhibit this phenomenon because we only represent the average of the executions.

7. Guiding the Constraint Solver

Constraint satisfaction is performed by splitting the domain of a variable into two or more parts and by propagating constraints [3]. It is generally well known that the performance of the solver depends upon the order in which variables are resolved. We choose to exploit this ‘flexibility’ for a variety of goals, namely, fairness, priority, avoiding no-flow solutions, and efficiency.

The following criteria, often used in conjunction, can help to guide the constraint solving process:

Variable ordering — Choose which variables to resolve first. For example, try evaluating variables corresponding to ends with data, such as from

external components or from a full FIFO_1 buffer, or select variable corresponding to high priority choices. Variables can be ordered locally, within a primitive to achieve a local notion of priority, or across parts of the connector or even globally across the entire connector to achieve a more global notion of priority by making the solver consider particular connectors before others.

Solution ordering — Choose the order in which that values are tried for each variable. For example, try solving constraints with a synchronisation variable set to \top before trying with \perp . Solution ordering can be applied to other data domains, though it is not immediately clear what the consequences of this would be.

We now describe how to use these to achieve certain effects. In general, the ordering imposed on variables is partial, leaving room for the constraint solver to makes its own choices.

Fairness — To implement nondeterministic choice fairly, the constraint solver needs to avoid resolving constraints in the same order each time it runs. Otherwise, it is possible that the same solution is always chosen. This can be achieved by randomising the variable ordering each time the constraint solver is invoked and/or changing the order in which the values of split variables are explored. In the presence of other constraints on the variable/value ordering, randomisation can occur modulo the imposed ordering.

Priority — Priority can be achieved by appropriately ordering the variables and/or the solutions to achieve the desired effect. The more global or comprehensive the ordering, that is, the more variables the ordering talks about, the more global the notion of priority. Purely local notions concerning as few as one variable are also sensible. For example, preferring flow over no-flow on the output end of a LossySync achieves a local preference for data flowing through the LossySync .

Avoiding no-flow solutions — To give priority of flow over no-flow, the solver is forced to try $x = \top$ before $x = \perp$ when resolving synchronisation variables $x \in \mathcal{X}$. This needs to be done for all variables corresponding to sources of data.

Efficiency — In general, the most efficient way to solve constraints is by starting with sources of data (such as inputs from components or full FIFO_1 buffers), and moving in the direction of data flow. Thus, the topology of the connector can also be used to help determine the variable ordering.

Example 9. *Consider again the Priority Merger channel from Table 3. It performs a merge of ends a and b into end c , giving priority to end a whenever both a and b are possible. The ordering constraints to achieve this consist of visiting variable a before visiting b , and then considering $a = \top$ before $a = \perp$.*

The exact degree to which the underlying constraint solver can be manipulated depends upon the implementation of the constraint solver. For instance, CHOCO,¹² provides some control, such as setting the order of values for a set of variable, though not to the extent described here. In other settings, the client of the constraint solver may have little influence over its internal algorithms. We also need to ensure that the various orderings are preserved by optimisations and by the composition of constraints. For example, if a variable is eliminated, what happens to the orderings related to the eliminated variable? More research is required to better understand this issue.

8. Implementing Interaction

Interaction between components and the engine in our model differs from previous descriptions of $\mathcal{R}eo$, as depicted in Figure 9. The usual interaction model for $\mathcal{R}eo$ components has two steps: firstly, a component attempts to write or take a data value; secondly, in the current or in some subsequent round the engine replies, with a possible data value. This is how $\mathcal{R}eo$ is implemented in Reolite [20] and in the current ECT toolkit.¹³



Figure 9: (Left) $\mathcal{R}eo$ -style interaction. (Right) Interaction in our approach.

In our model, components play a more participatory role, wherein they publish a ‘meta-level’ description of their possible behaviour in the current round in the form of a constraint. The engine replies with a term that the component interprets as designating its new state and, if required, the data flow that occurred. This is (a part of) the solution to the constraints, typically just the value assigned to the $state'_C$ variable.

The new (interactive) approach can easily be wrapped to look like the previous (non-interactive) approach as follows, so that components written for an older version of the $\mathcal{R}eo$ engine can be used with this version:

write: Component C issues a write of data d to end a :

1. Pass the constraint $a \rightarrow (\hat{a} = d \wedge state'_C = \text{ok}) \wedge \neg a \rightarrow state'_C = \text{no}$ to the constraint solver.
2. If the constraint solver returns $state'_C = \text{ok}$, return control to the component.

¹²CHOCO constraint programming system, available from <http://choco.sourceforge.net/>

¹³<http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools>

3. Otherwise, try again with the same constraint in the next solver round.

take: Component C issues take on end a satisfying constraint $P(x)$:

1. Pass the constraint $a \rightarrow (P(\hat{a}) \wedge state'_C = \text{ok}(\hat{a})) \wedge \neg a \rightarrow state'_C = \text{no}$ to the constraint solver.
2. If constraint solver returns $state'_C = \text{ok}(\hat{a})$, return control with value \hat{a} to the component.
3. Otherwise, try again with the same constraint in the next solver round.

In general, the interaction protocol between the constraint solver and the component consists of the component/primitive issuing constraints to the solver over a certain set of variables and the solver returning the values of those variables to the component/primitive. Typically, it is sufficient to encode the information returned by the constraint solver in the value stored in the state variable.

9. Contribution to \mathcal{Reo}

The surface syntax of \mathcal{Reo} is presented in terms of channels and their connecting nodes. In order to offer intuition as to how \mathcal{Reo} works, analogies have been drawn between the behaviour of a connector and the flow of electricity in an electrical circuits or water flow through pipes. Such systems have a natural equilibrium-based realisation, which does not extend to data flow in \mathcal{Reo} , and this becomes apparent when trying to implement \mathcal{Reo} . Indeed, even describing \mathcal{Reo} purely in terms of data flow can be misleading, as the direct approach to implementing \mathcal{Reo} , by plugging together channels and having them locally pass on data according to the channel's local behavioural constraints, does not work. The channel abstraction offers no help as channels are only capable of locally deciding what to do, whereas the behaviour of a connector typically cannot be locally determined: *It is impossible to make choices that are local to channels in order to satisfy the constraints imposed by the entire connector.* This means, for example, that a proposed implementation based on the MoCha middleware [29], which has all the primitive channels \mathcal{Reo} has and nothing else, *cannot* possibly work without additional infrastructure. (This was already identified by Guillen Scholten [29].) Specifically, some form of backtracking or non-local arbitration would be required to guarantee the atomicity between the sending of data and the receiving of data, while obeying the constraints imposed by the connector. An attempt to provide a distributed model for \mathcal{Reo} based on the speculative passing of data combined with back-tracking has been made [24], but the result was too complex, possibly because it too closely followed the channel metaphor.

To see why, consider the connector in Figure 10. End x could speculatively send data through both the FIFO buffer ($x-y$) (satisfying its local constraints) and through the Sync channel ($x-v$). Node v would then send the data through the Sync channel ($v-w$). Node w must also send data into the SyncDrain ($w-z$),

but no data will ever arrive at end z (in this round), so the SyncDrain cannot synchronise. Thus the constraints of all the primitives are not satisfied, based on the wrong initial choice at x , so the entire data flow needs to be rolled back. In a distributed setting, this is unlikely to be a feasible approach to implementing $\mathcal{R}eo$.

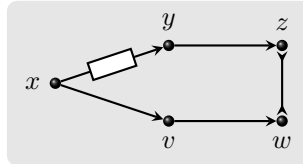


Figure 10: Example $\mathcal{R}eo$ connector illustrating the need for global choices.

Although $\mathcal{R}eo$ is described in terms of channels and their connecting nodes, existing $\mathcal{R}eo$ models are instead based on four main concepts: synchronisation, data flow, state transition, and context dependency. Every model of $\mathcal{R}eo$ focusses on one or more of these concepts. All more recent attempts to implement $\mathcal{R}eo$ are based directly on one of its formal semantic models. This means that the limitations of a semantic model are inherited by implementations based on that model—primitives *cannot* offer arbitrary behavioural possibilities, but are restricted by the semantic model underlying the implementation—, but in general it gives considerable freedom in choosing an implementation approach.

We therefore present a comparative discussion of some of these models and their implications for the implementation of $\mathcal{R}eo$. We break our discussion into two parts. Firstly, we discuss existing models of $\mathcal{R}eo$. Then we compare existing implementation approaches, including some failed attempts.

9.1. $\mathcal{R}eo$ Models

Models of $\mathcal{R}eo$ try to capture one or more of the following features: synchronisation, data flow, state, context dependency, and, more recently, reconfiguration. Some models aim to be comprehensive, covering as many features as possible, whereas others focus on one or two in order to better understand certain issues.

Synchronisation corresponds to two or more actions occurring atomically; it is the only notion common to all $\mathcal{R}eo$ models. We consider mutual exclusion or asynchrony, namely, expressing that two actions cannot occur together, as falling under the umbrella of synchronisation.

Data aware models describe the values of data being communicated, as well as permitting synchronisation that depends upon the value being sent. For example, synchronisation at the sink end of a filter channel, depends upon the value passed through its source end. Models not mentioning data can still be implemented to pass data, but not to transform it nor base synchronisation upon it.

Context dependency is a notion required to model behaviour that changes non-monotonically with the context in which the connector is placed. For example, the original intended behaviour of a LossySync channel is that it will lose data only if the primitive connected to its sink end does not accept the data [4]. Related to context dependency are the notions of priority, which prefers one transition (in an automata model) over another whenever both are possible, and maximal flow, which prefers data to spread as far as possible into a connector (all other things being equal) [42]. Different approaches have been explored, as it is unclear from the informal descriptions of $\mathcal{R}eo$ what exactly context dependency should mean, in particular, regarding how it should interact with non-determinism.

Reconfiguration occurs when channels are unplugged from each other and plugged together in a new configuration. This can be initiated from within a connector or as an external action. We have said nothing about reconfiguration before, but recent work proposes a number of approaches to it [19, 34]. For this discussion we are only interested in the impact that facilitating reconfiguration has on determining the behaviour of a connector.

Abstract Behaviour Types. The original semantics of $\mathcal{R}eo$ is defined in terms of abstract behaviour types [5], which are co-inductively defined relations over timed data streams (a time stream paired with a value stream). These models account for synchronisation, data flow, and, implicitly, state changes. They mention neither context dependency nor reconfiguration.

This semantics provide little guidance for implementing $\mathcal{R}eo$, so we do not consider it further.

Automata-based Models. Numerous automata-based models for $\mathcal{R}eo$ exist. We consider the following four: constraint automata (CA) [14], intensional constraint automata (ICA) [23], port automata (PA) [33], and $\mathcal{R}eo$ automata (RA) [17]. These models provide the semantics of each $\mathcal{R}eo$ primitive and their composition, by representing the synchronisation possible in a connector and possibly a description of the data flow in the transitions of the automata. In all of these models synchronisation is represented by a set of ends in each transition, and the data flow by constraints.

CA is the only automata model of $\mathcal{R}eo$ that captures data, as the other models focus on different issues. PA were devised to study decomposition of automata, and hence they capture only synchronisation and state. Neither CA nor PA have context dependency, so they can only express a variant of LossySync which makes a non-deterministic choice between passing the data onwards and losing it. Context dependency was later incorporated in an automata model, firstly in ICA, by reverse engineering ideas from connector colouring, and, subsequently and more compactly, in RA.

Reconfiguration in the presence of an automata-based model requires both keeping a description of the connector around and completely re-computing the underlying semantics of a connector whenever reconfiguration occurs [19].

Additional variants of constraint automata proposed as \mathcal{Reo} models include notions of time [7], quality-of-service guarantees [9], resource bounds [37], or probability [15].

Connector Colouring. *Connector colouring* [20] is based on the simple idea that ends in a connector where data flows and where data does not flow can be coloured with different colours (see Section 5). Two variants of connector colouring exist. The first, called 2-colouring (CC2), has one colour representing data flow and one representing no data flow. The second, called 3-colouring (CC3), splits the no data flow colour into two to capture context, as described in Section 5. CC2 captures only synchronisation, whereas CC3 captures both synchronisation and context dependency. Both connector colouring schemes abstract away from the data passed and state transition, but this information can be added in implementations, as long as the value of the data does not affect synchronisation and it is not transformed.

As connector colouring is computed on-the-fly, reconfiguration has no impact on the computation of connector semantics, as the most recent version of the connector is used to compute the colouring table.

SOS. Mousavi *et. al.* present a structural operational semantics (SOS) formalisation of \mathcal{Reo} using Maude [42]. Two versions of the semantics were developed: the original formulation (SOS) and an extension with a notion of maximal flow to capture a notion of context dependency (SOS+FLOW). Khosravi *et. al.* further explore this idea using Alloy [30]. One advantage of the SOS approach is that it deals with causality issues in connectors (which we deliberately ignore, as most other models of \mathcal{Reo} do). Clarke [18] explores causality in depth, presenting \mathcal{Reo} semantics in terms of the proof theory of intuitionistic linear logic and zero-safe Petri nets. These approaches do not consider reconfiguration.

An alternative operational semantics is based on the Tile Model [8]. This approach extends connector colouring (both 2- and 3-colouring) to include data, state and a primitive notion of reconfiguration (though not causality). As such, the tile models (TILE2 and TILE3) are the most complete semantic descriptions of \mathcal{Reo} .

Constraint-based Approach. Our constraint approach deals with synchronisation, data awareness, state, and context dependency in an orthogonal and uniform way. We defined different constraints for each of these notions, which are then added conjunctively to capture any combination of them (along with certain axioms connect the various kinds of variables). Beyond the other models, constraints can be used to express that multiple inputs are available on a particular node, whereas other models deal only with a single datum (or do not mention data at all).

Reconfiguration was not considered for the model presented in this paper, though it would be implemented by rewriting the appropriate constraints.

Figure 11 presents a comparison of the various approaches along four of the dimensions of interest (all models express synchronisation). Only three of

Model	Data Awareness	State	Context Dependency	Reconfiguration
ABT	✓	✓	✗	None
CA	✓	✓	✗	Re-compute
ICA	✗	✓	✓	Re-compute
PA	✗	✓	✗	Re-compute
RA	✗	✓	✓	Re-compute
CC2	✗	✗	✗	Compatible
CC3	✗	✗	✓	Compatible
SOS	✓	✓	✗	Compatible
SOS+FLOW	✓	✓	✓	Compatible
TILE2	✓	✓	✗	Some
TILE3	✓	✓	✓	Some
Constraints	✓	✓	✓	Compatible

Figure 11: Comparison of $\mathcal{R}eo$ models. *None* means that reconfiguration is unfeasible. *Re-compute* means that reconfiguration would require re-computing the entire semantics. *Compatible* means that the model is compatible with reconfiguration, because semantics are computed on-the-fly. *Some* means that the model has some notion of reconfiguration built in.

the models receive three ✓s, namely SOS+FLOW, TILE3, and Constraints. SOS+FLOW was not considered for implementation as the notion of maximal flow is not as flexible as the notion of context dependency enforced by 3-colouring. As TILE3 extends the 3-colouring model to include data awareness and state, and we have shown how to encode synchronisation, data awareness, state, and 3-colouring style context dependency into constraints, we conclude that these are semantically equivalent approaches.

9.2. $\mathcal{R}eo$ Engines

The coordination abstractions provided by $\mathcal{R}eo$ impose some implementation challenges, so most approaches to implementing $\mathcal{R}eo$ involve directly implementing some semantic model. Each semantic model forces different characteristics and limitations on implementations based on it. We now compare existing and possible implementations of $\mathcal{R}eo$ on the following points:

implementation approach The approaches we cover include a speculative approach, compilation into automata, connector colouring, a search-based approach, and constraint solving.

number of solutions computed when determining what to do in the next step, two approaches are possible:

1. (**all-sol**) find all possibilities for a round, and choose one of them non-deterministically (or based on some other scheme); and
2. (**one-sol**) find only some (typically one) of the possible solutions.

pre-computed behaviour when computing the behaviour of a connector, two approaches are possible:

1. (**all-steps**) pre-compute all future behaviour *a priori*; and
2. (**single-step**) compute the behaviour of a single step at a time.

Speculative Approach. This approach to implementing \mathcal{Reo} consists of speculatively trying to send data through channels and rolling back when an inconsistency arose. Had such an approach been successfully implemented, it would have computed one solution at a time (**one-sol**) for a single step (**single-step**). This is all speculation, however, as this approach was never successfully implemented due to the inherent impossibility of locally making the globally consistent choices, and the difficulty of managing distributed rollback and subsequent retries.

Automata-based Implementations. This approach to implementing \mathcal{Reo} compiles the behaviour of a connector into an automaton [36], and thus pre-computes all future behaviour at compile-time (**all-steps**, **all-sol**). Implementations of all-steps semantics do not scale, since finding all possible behaviour for all possible states of a concurrent system is very expensive and space inefficient. For example, the number of states generally doubles for every FIFO1 buffer in a connector, assuming all states are reachable. Certainly, infinite state spaces are excluded. Furthermore, implementations based on automata models are inherently centralised, and lose all potential parallelism, as a connector is implemented using a single automaton. On the other hand, this means that they can be efficient at run-time, though they need to be re-computed when reconfiguration occurs. They trade off run-time efficiency for flexibility.

Connector Colouring-based Implementations. These encode the behaviour of the next step of a connector as a colouring table and compose the colouring tables using a notion of matching, as described above in Section 5. Implementations based on this approach compute all solutions (**all-sol**) for a single step (**single-step**). The disadvantage of computing all steps is the potential overhead of computing choices that are not used.

Reolite [20] was the first prototype implementation based on connector colouring. A more recent and efficient engine is incorporated into the \mathcal{Reo} toolset [10]. The latter version was used as a comparison in our benchmarking. Connector colouring also forms the basis of our upcoming distributed implementation, which we discuss below in Section 9.3.

Search-based Implementation. These include implementations based on the SOS models and, hypothetically, on the Tile models. The Maude implementation of SOS [42] could find a single solution (**one-sol**), whereas the implementation of SOS+FLOW [42] computed all possibilities (**all-sol**), which were then ordered based on a maximal flow test. The result was extremely inefficient, although the encoding of SOS+FLOW in Alloy [30] potentially offers a better **one-sol** implementation technique using SAT solving. Unfortunately, no benchmarks

were presented in that paper, and as the implementation is very much a prototype, we have not included it here for comparison. TILE2 and TILE3 could easily be implemented in a similar fashion using Maude, though this has not been done, as far as we are aware.

Constraint Satisfaction. The implementation approach described in this paper is based on constraint satisfaction techniques to derive efficient executable implementations of *Reo*. The constraints are solved per round (**single-step**) and use the heuristics of the constraint solver to stop the search for solutions once a single solution is found, avoiding exploring the full solution space (**one-sol**).

We summarise the classification of implementation approaches discussed in this section in Table 8.

Implementation Approach	Number of Solutions	Pre-computed Behaviour
Speculative approach	one	single
Compilation into automata	all	all
Connector colouring	all	single
Search-based	one	single
Constraint satisfaction	one	single

Table 8: Classification of *Reo* implementations approaches.

9.3. A Distributed Implementation of *Reo*

Prior to this work, we developed a prototype distributed implementation of *Reo*, where we restrict communication so that it can occur only through primitives, and thereby prohibiting both a global agent and direct node-to-node communication. This restriction imposes additional obligations on the implementation of primitives. Specifically, it requires them to play a significant role in the global constraint resolution process, for instance, to pass around colouring tables and to serve as the conduits for all “coordination communication,” as well as for normal communication.

In the distributed implementation the primitives follow a distributed protocol to achieve a consensus regarding how data should flow in each round, and only then data is passed through the primitives. Our prototype implementation relies on connector colouring, but we now are incorporating constraint-solving techniques to improve efficiency. This implementation is ongoing work that has not yet been published, but more information is available online.¹⁴ (It will be reported in the second author’s forthcoming Ph.D. dissertation.)

¹⁴Available from <http://reo.project.cwi.nl/>.

10. Related work

Wegner describes coordination as constrained interaction [51]. As such, coordination systems can be modelled by *interaction machines*. Interaction machines react to real-time interactive behaviour, representing the external world by infinite streams of inputs, allowing them to go beyond Turing machines in expressive power. The implementation model of \mathcal{Reo} presented in this paper, which is extended with an interaction layer, can be regarded as a concrete realisation of Wegner’s interaction machine.

However, surprisingly little work takes Wegner’s view of coordination as constrained interaction literally, representing coordination as constraints. Montanari and Rossi express coordination as a constraint satisfaction problem, in a similar but more general way [41]. They describe how to solve synchronisation problems using constraint solving techniques. Networks are viewed as graphs, and the tile model is used to distinguish between synchronisation and sequential composition of the coordination pieces. In our approach, we clarify one possible semantics of the coordination language \mathcal{Reo} in these terms, giving a clear meaning for each variable, and describing the interaction with the external world within the solve and update stages. Lazovik *et al.* also utilise constraints to solve a coordination problem [35]. They provide a choreography framework for web services, where choreography is formalised as a constraint programming task, and where both the Business Process and the requests are modelled as a set of constraints. This is an example of a concrete application of constraints to coordination, using a centralised and non-compositional approach.

Taking a more practical approach, Minsky and Ungureanu introduce the Law-Governed Interaction (LGI) mechanism [40], implemented by the Moses toolkit. This mechanism targets distributed coordination of heterogeneous agents using a policy that enforces extensible laws. Laws are constraints specified in a Prolog-like language, enforced on regulated events of the agents, such as send or receive of messages. The authors give a special emphasis to the deployment and execution of the mechanism, where a trusted server provides certified controllers which enforce the laws, instead of relying on a centralised coordination mechanism. However, laws are local, in the sense that can only refer to the agent being regulated. This allows them to achieve good performance using LGI. In the presence of true global constraints, as in \mathcal{Reo} , LGI would require more complex algorithms.

Frølund [27] presents *synchronisers* as a part of an actor coordination framework. He gives semantics for these constructs in terms of constraints, but does not use constraint solving as an implementation technique. The constraints perform the matching of atomic sets of actions along with pattern matching of data, but they do not deal with the communication of data, as our model does. Frølund’s synchronisers cannot be plugged together like channels, but they can be composed by overlapping the domains of multiple synchronisers.

The analogy between \mathcal{Reo} constraints and constraint solving problems has already been made in general publications about \mathcal{Reo} [6]. Since then more specific approaches that utilise a constraint-based perspective over \mathcal{Reo} have

been proposed. Examples of these approaches look at model checking and at the use of mashups. Klüppelholz and Baier describe a symbolic model checking approach for $\mathcal{R}eo$ [32]. Constraint automata are represented by binary decision diagrams, encoded as propositional formulæ. Their encoding is similar to ours, though they use exclusively boolean variables, whilst we deal with a richer data domain. Maraïkar *et al.* [36] present a service composition platform based on $\mathcal{R}eo$, adopting a mashup’s data-centric approach. They combine several RSS feeds into a user interface using a $\mathcal{R}eo$ connector, that is executed using the CHOCO constraint solver (referred in Sections 7 and 6). The work by Maraïkar *et al.* can be seen as an example of an application of the basic ideas that we present in this paper.

The timed concurrent constraint (tcc) programming framework [47] was introduced by Saraswat *et al.* to integrate the concurrent constraint (cc) programming paradigm [46] with synchronous languages. Time units are rounds, all the constraints are updated in each round, as ours are, whereas inside each round the constraints are computed to quiescence. cc programs are compiled into an automata model, where states are cc programs and transitions represent evolution *within a round* while solving the constraints. In contrast, transitions in the constraint automata model for $\mathcal{R}eo$ describe the evolution between rounds. Furthermore, the tcc approach avoids non-determinism as it targets synchronous languages, whilst $\mathcal{R}eo$, as a coordination language, embraces non-determinism.

Andreoli *et al.* [2, 1] also use (linear) logic as the basis for coordination, combining it with the object-oriented paradigm. They utilise proof search to reason about coordination, as opposed to the use of constraint satisfaction techniques to derive efficient implementations. Clarke follows a similar approach and presents a $\mathcal{R}eo$ semantics, also based on proof search, using an extension of linear logic with temporal modalities in an intuitionistic setting [18].

A small overview of other coordination models, and its relation with $\mathcal{R}eo$, is now in order. The survey of Papadopoulos and Arbab [43] compares several coordination languages, classifying languages based on tuple spaces as data-driven models, as opposed to $\mathcal{R}eo$ ’s channel-driven model. Manifold [12] is another example of a channel-driven model presented in the survey, upon which $\mathcal{R}eo$ was built. Linda [28], one of the first coordination languages, provides a simple executable model consisting of a shared tuple space that components use to exchange values. Several other variations, such as Java’s popular implementation JavaSpace of Jini [26], and the Klaim language [16], which considers multiple distributed tuple spaces, followed the basic ideas behind Linda. The foundations of Klaim are presented as a process calculus, in particular as a variant of the π -calculus [39] with process distribution and mobility, where communication occurs via shared located repositories instead of channel-based communication primitives. Individual tuple operations in Linda-like languages are atomic, though they do not provide the global synchronisation imposed by $\mathcal{R}eo$.

Coordination languages such as SRML [25] and Orc [31] are oriented towards the coordination of web-services. SRML is a language developed in the context of the SENSORIA project, where the interaction between two component services can be either synchronous or asynchronous, depending on whether

an acknowledgement is required or not. Orc assumes a centralised coordinator that communicate with the component services only via asynchronous messages, instead of describing one to one communication. It also assumes that each service can reply at most once. *Reo* takes the same exogenous approach as Orc, moving the coordination logic from the components to the coordinator, and introduces new synchronisation capabilities, not captured by the Orc language. An exhaustive formal comparison between *Reo* and Orc was performed by Proença and Clarke [44].

Coordination models have been applied to coordinate solvers of distributed constraint satisfaction problems (DCSP) [12]. Our coordination model comes full circle and is based on (D)CSP.

11. Conclusion and Future Work

We presented a new semantic and executable model for the *Reo* coordination model based on constraint satisfaction. This was motivated by one main concern. The existing models of *Reo* lack an efficient implementation technique. The channel-view of a *Reo* connector becomes a mere metaphor. Instead, a *Reo* connector is seen as a set of constraints, based on the way the primitives are connected together, and their current state, governing the possible synchronisation and data flow at the channel ends connected to external entities. The circuit representation of a *Reo* connector then serves as a convenient graphical representation of how primitive constraints compose to yield the constrained interaction that it expresses.

We contribute the state-of-the-art of *Reo* in two ways:

- We identify the four main concepts that identify the *Reo* coordination, synchrony, data-awareness, state, and context dependency, and describe these concepts using logical constraints. We show the correctness of our approach with respect to the first three concepts using the constraint automata model as a reference, and we give a full proof of correctness for the latter concept based on the connector colouring semantics.
- We reuse existing constraint satisfaction techniques to derive more efficient implementations of *Reo*. Specifically, we developed two prototypes implementations, one which uses a SAT solver and another which uses a constraint solver to search for possible solutions, and compared their performance with an existing *Reo* engine based on connector colouring. The results strongly support the idea that constraint solving can be a good approach for implementing coordination languages.

In a previous version of this work, the authors explored the decomposition of *Reo* into constraints, extended the framework presented here to model interaction with an unknown external world [22], beyond what is currently possible in existing implementations of *Reo*. In that paper we assume that parts of the constraints are still unknown in the beginning of the constraint solving phase. The corresponding constraints could be requested from external entities when

required. Clarke and Proença introduced a *local logic* [21], wherein constraints from only part of a connector need be consulted when searching for valid solutions. This means that partial solutions over only some of the variables are admitted, making this approach more scalable. Furthermore, partiality allows one to reason about ‘incomplete’ constraints, which can be extended during the constraint satisfaction process, enabling a new model of interaction with the external world.

Ongoing work on *Reo* tools and on the distributed engine of *Reo* can benefit from more comprehensively supporting the model proposed here. In particular, the distributed engine currently uses the connector colouring semantics. This means that data constraints are beyond the model, and the only interaction allowed is writing and reading on the channel ends, as explained in Section 8, which can be improved using the constraints-based model proposed in this paper. Constraints provide a uniform and flexible framework in which it may be possible in the future to combine with other constraint based notions, such as service-level agreements. Future work will explore these directions, in particular, the increased expressiveness offered by constraints and the external interaction modes the model offers. In addition, we will also try to exploit the parallelism inherent in constraints.

References

- [1] Andreoli, J.-M., Freeman, S., Pareschi, R., 1996. The coordination language facility: coordination of distributed objects. *Theory and Practice of Object Systems* 2 (2), 77–94.
- [2] Andreoli, J.-M., Pareschi, R., 1990. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, 495–510.
- [3] Apt, K., 2003. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA.
- [4] Arbab, F., 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14 (3), 329–366.
- [5] Arbab, F., 2005. Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming* 55 (1-3), 3–52.
- [6] Arbab, F., 2006. Composition of interacting computations. In: Goldin, D., Smolka, S., Wegner, P. (Eds.), *Interactive Computation: The New Paradigm*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, pp. 277–321.
- [7] Arbab, F., Baier, C., de Boer, F. S., Rutten, J. J. M. M., 2007. Models and temporal logical specifications for timed component connectors. *Software and System Modeling* 6 (1), 59–82.

- [8] Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U., 2008. Tiles for Reo. In: Proceedings of Recent Trends in Algebraic Development Techniques (WADT). Vol. 5486 of Lecture Notes In Computer Science. Springer-Verlag, Berlin, Heidelberg, pp. 37–55.
- [9] Arbab, F., Chothia, T., Meng, S., Moon, Y.-J., 2007. Component connectors with QoS guarantees. In: Murphy, A. L., Vitek, J. (Eds.), COORDINATION. Vol. 4467 of Lecture Notes in Computer Science. Springer, pp. 286–304.
- [10] Arbab, F., Koehler, C., Maraikar, Z., Moon, Y., Proença, J., 2008. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In: International Workshop on Formal Aspects of Component Software (FACS). Electronic Notes in Theoretical Computer Science (ENTCS), Malaga.
- [11] Arbab, F., Kokash, N., Meng, S., 2008. Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (Eds.), ISoLA. Vol. 17 of Communications in Computer and Information Science. Springer, pp. 108–123.
- [12] Arbab, F., Monfroy, E., 1998. Coordination of heterogeneous distributed cooperative constraint solving. SIGAPP Applied Computing Review 6 (2), 4–17.
- [13] Arbab, F., Sun, M., Baier, C., 2009. Synthesis of Reo circuits from scenario-based specifications. Electronic Notes in Theoretical Computer Science 229 (2), 21–41.
- [14] Baier, C., Sirjani, M., Arbab, F., Rutten, J., 2006. Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61 (2), 75–113.
- [15] Baier, C., Wolf, V., 2006. Stochastic reasoning about channel-based component connectors. In: Ciancarini, P., Wiklicky, H. (Eds.), COORDINATION. Vol. 4038 of Lecture Notes in Computer Science. Springer, pp. 1–15.
- [16] Bettini, L., Bono, V., Nicola, R. D., Ferrari, G., Gorla, D., Loreti, M., Moggi, E., Pugliese, R., Tuosto, E., Venneri, B., 2003. The Klaim project: Theory and practice. In: Global Computing: Programming Environments, Languages, Security and Analysis of Systems. Vol. 2874 of Lecture Notes in Computer Science. Springer-Verlag, pp. 88–150.
- [17] Bonsangue, M. M., Clarke, D., Silva, A., 2009. Automata for context-dependent connectors. In: Field, J., Vasconcelos, V. T. (Eds.), COORDINATION. Vol. 5521 of Lecture Notes in Computer Science. Springer, pp. 184–203.

- [18] Clarke, D., 2007. Coordination: Reo, nets, and logic. In: de Boer, F. S., Bonsangue, M. M., Graf, S., de Roever, W. P. (Eds.), FMCO. Vol. 5382 of Lecture Notes in Computer Science. Springer, pp. 226–256.
- [19] Clarke, D., 2008. A basic logic for reasoning about connector reconfiguration. *Fundamenta Informaticae* 82 (4), 361–390.
- [20] Clarke, D., Costa, D., Arbab, F., May 2007. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming* 66 (3), 205–225.
- [21] Clarke, D., Proença, J., 2009. Coordination via interaction constraints i: Local logic. CoRR abs/0911.5445.
- [22] Clarke, D., Proença, J., Lazovik, A., Arbab, F., 2009. Deconstructing Reo. *Electronic Notes in Theoretical Computer Science* 229 (2), 43–58.
- [23] Costa, D., 2010. Formal models for context dependent connectors for distributed software components and services. Ph.D. thesis, to appear.
- [24] Everaars, C. T. H., de Oliveira Costa, D. F., Diakov, N. K., Arbab, F., February 2006. A distributed computational model for Reo. Tech. Rep. SEN-E0601, CWI, Amsterdam, The Netherlands.
- [25] Fiadeiro, J. L., Lopes, A., Bocchi, L., 2006. A formal approach to service component architecture. In: Bravetti, M., Núñez, M., Zavattaro, G. (Eds.), WS-FM. Vol. 4184 of Lecture Notes in Computer Science. Springer, pp. 193–213.
- [26] Freeman, E., Arnold, K., Hupfer, S., 1999. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK.
- [27] Frølund, S., 1996. *Coordinating Distributed Objects*. The MIT Press, Cambridge, Massachusetts, USA.
- [28] Gelernter, D., 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7 (1), 80–112.
- [29] Guillen Scholten, J. V., January 2007. Mobile channels for exogenous coordination of distributed systems : semantics, implementation and composition. Ph.D. thesis, LIACS, Faculty of Mathematics and Natural Sciences, Leiden University.
- [30] Khosravi, R., Sirjani, M., Asoudeh, N., Sahebi, S., Iravanchi, H., 2008. Modeling and analysis of Reo connectors using Alloy. In: Lea, D., Zavattaro, G. (Eds.), COORDINATION. Vol. 5052 of Lecture Notes in Computer Science. Springer, pp. 169–183.
- [31] Kitchin, D., Cook, W. R., Misra, J., 2006. A language for task orchestration and its semantic properties. In: CONCUR. pp. 477–491.

- [32] Klüppelholz, S., Baier, C., 2007. Symbolic model checking for channel-based component connectors. *Electronic Notes in Theoretical Computer Science* 175 (2), 19–37.
- [33] Koehler, C., Clarke, D., 2009. Decomposing port automata. In: SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing. ACM, New York, NY, USA, pp. 1369–1373.
- [34] Koehler, C., Lazovik, A., Arbab, F., 2008. Connector rewriting with high-level replacement systems. *Electronic Notes in Theoretical Computer Science* 194 (4), 77–92.
- [35] Lazovik, A., Aiello, M., Gennari, R., 2006. Choreographies: Using constraints to satisfy service requests. In: AICT-ICIW '06: Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services. IEEE Computer Society, Washington, DC, USA, p. 150.
- [36] Maraïkar, Z., Lazovik, A., Arbab, F., 2008. Building mashups for the enterprise with SABRE. In: Bouguettaya, A., Krüger, I., Margaria, T. (Eds.), ICSOC. Vol. 5364 of *Lecture Notes in Computer Science*. pp. 70–83.
- [37] Meng, S., Arbab, F., 2007. On resource-sensitive timed component connectors. In: Bonsangue, M. M., Johnsen, E. B. (Eds.), FMOODS. Vol. 4468 of *Lecture Notes in Computer Science*. Springer, pp. 301–316.
- [38] Meng, S., Arbab, F., 2007. Web services choreography and orchestration in Reo and constraint automata. In: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing. ACM, New York, NY, USA, pp. 346–353.
- [39] Milner, R., June 1999. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press.
- [40] Minsky, N. H., Ungureanu, V., 2000. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology* 9 (3), 273–305.
- [41] Montanari, U., Rossi, F., 1998. Modeling process coordination via tiles, graphs, and constraints. In: 3rd Biennial World Conference on Integrated Design and Process Technology. Vol. 4. pp. 1–8.
- [42] Mousavi, M. R., Sirjani, M., Arbab, F., 2006. Formal semantics and analysis of component connectors in Reo. *Electronic Notes in Theoretical Computer Science* 154 (1), 83–99.
- [43] Papadopoulos, G. A., Arbab, F., 1998. Coordination models and languages. In: M. Zelkowitz (Ed.), *The Engineering of Large Systems*. Vol. 46 of *Advances in Computers*. Academic Press, pp. 329–400.

- [44] Proenca, J., Clarke, D., 2007. Coordination Models Orc And Reo Compared. In: Proceedings of the International Workshop on the Foundations of Coordination Languages and Software Architecture (FOCLASA). Elsevier.
- [45] Rutten, J. J. M. M., 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249 (1), 3–80.
- [46] Saraswat, V. A., 1993. Concurrent constraint programming. MIT Press, Cambridge, MA, USA.
- [47] Saraswat, V. A., Jagadeesan, R., Gupta, V., Nov.–Dec. 1996. Timed default concurrent constraint programming. *Journal of Symbolic Computation* 22 (5–6), 475–520, extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.
- [48] Sheini, H. M., Sakallah, K. A., 2006. From propositional satisfiability to satisfiability modulo theories. In: Biere, A., Gomes, C. P. (Eds.), SAT. Vol. 4121 of Lecture Notes in Computer Science. Springer, pp. 1–9.
- [49] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., Barros, A. P., 2003. Workflow patterns. *Distributed and Parallel Databases* 14 (1), 5–51.
- [50] Venkataraman, K. N., 1987. Decidability of the purely existential fragment of the theory of term algebras. *J. ACM* 34 (2), 492–510.
- [51] Wegner, P., 1996. Coordination as constrained interaction (extended abstract). In: *Coordination Languages and Models*. Vol. 1061 of Lecture Notes in Computer Sciences. pp. 28–33.
- [52] Xie, M., 2005. Specification Of E-Business Process Model For PayPal Online Payment Process Using Reo. Master’s thesis, Leiden University, the Netherlands.