Partial Connector Colouring*

Dave Clarke and José Proença

IBBT-DistriNet, Department of Computer Science,
K.U.Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract. Connector colouring provided an intuitive semantics of Reo connectors which lead to effective implementation techniques, first based on computing colouring tables directly, and later on encodings of colouring into constraints. One weakness of the framework is that it operates globally, giving a colouring to all primitives of the connector in lock-step, including those not involved in the interaction. This global approach limits both scalability and the available concurrency. This paper addresses these problems by introducing partiality into the connector colouring model. Partial colourings allow parts of a connector to operate independently and in isolation, increasing scalability and concurrency.

1 Introduction

Reo [1] is a visual language for coordinating components and web services using connectors composed from a small, but open, collection of primitives. A lot of research has gone into semantic models for Reo, with the following (incomplete) goals: to facilitate reasoning about Reo connectors; to provide context-dependent behaviour; and to enable efficient implementations. Connector colouring [8] provided a great leap forward on these three fronts. Firstly, it is an intensional semantic model that also gives a visual representation of what's going on inside a connector. Secondly, connector colouring offered the first context dependent semantics for Reo. Thirdly, the semantics were simple enough to enable an straightforward implementation of Reo [2]. Further improvements in implementation efficiency were obtained by encoding connector colouring as constraints and using SAT and constraint satisfaction techniques [9].

The problem with the resulting implementation approach, and indeed with connector colouring as a semantic model, is that it gives only a *global* description of the behaviour of a connector, with *all* primitives participating *lock-step* to determine what to do next. In the constraints encoding, the constraints of all primitives are conjoined together to compute the next step. This has three negative consequences: the amount of available concurrency is reduced; implementations are inherently unscalable; and the behavioural possibilities offered by connectors are actually reduced—some desirable behaviour becomes impossible.

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu) and KULeuven BOF Project STRT1/09/031: DesignerTypeLab.

This paper proposes a solution to this problem which overcomes these negative consequences, simply by considering certain partial connector colourings as valid. Rather than giving behaviour to the entire connector, a partial colouring can give behaviour to a part of a connector in a coherent fashion. This enables the local colouring of multiple parts of a connector concurrently and independently. The details of partial connector colouring are worked out for existing colouring schemes and new encodings as constraints are presented. We describe how existing constraint satisfaction engines can be used to implement partial colouring, and provide some benchmarks.

The paper is organised as follows. Section 2 gives a review of Reo and the connector colouring model. Section 3 details the problem being addressed. Section 4 describes one solution, namely, partial connector colouring. Section 5 reviews and adapts the constraint-based encoding of connector colouring. Section 6 discusses the implementation of our approach and gives some benchmarks. Section 7 discusses related work and Section 8 concludes.

2 Background: Reo and Connector Colouring

2.1 Reo Coordination Model

Reo [1] is a coordination model, wherein coordinating *connectors* are constructed by composing more primitive connectors. Reo's primitives, such as channels, offer a variety of behavioural policies regarding synchronisation, buffering, and lossiness. 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 intuition about the behaviour of a connector.

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 notions of source and sink in directed graphs.) Connectors are formed by plugging the ends of primitives together, without loss of generality, 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—the two ends in a node are synchronised. The behaviour of each primitive depends upon its current state and the semantics of a connector can 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*. Components attach to the boundary of a connector.

Some Reo primitives are:

Replicator $\left(a \longrightarrow \begin{smallmatrix} b \\ c \end{smallmatrix} \right)$ replicates data synchronously from a to b and c.

Merger copies data synchronously either from a to c or b to c, but not from both.

Priority merger behaves like a merger, but prefers to select data from a over b, if both alternatives are possible. The ! marks the higher priority input.

LossySync has two possible behaviours. Data can either flow synchronously from a to b, when possible, or can flow on a but not on b. The LossySync is context-dependent, in that it prefers to allow data to flow from a to b rather than lose it.

Sync passes data synchronously from a to b

SyncDrain synchronises ends a and b, consuming the data.

AsyncDrain consumes data either from a or from b, but not from both.

FIFO$_1$ is a stateful channel representing a buffer of size 1. When the buffer is empty it can receive data on a, but not output data on b. It then changes state and becomes full. A full FIFO$_1$ with data d, denoted, can output d through b, but cannot receive data on a. It then changes state and becomes empty again.

Exclusive Router (⊗) denotes an exclusive router, which has one input and multiple outputs. The input data is synchronously sent to exactly one output.

Doing nothing—no data flow—is always one of the behavioural possibilities.

In diagrams of connectors, nodes will be denoted (•). In general, these can also have multiple source and sink ends, to denote generalised mergers and replicators, but these can be encoded in terms of binary mergers and replicators.

Example 1. The *synchronous merge* connector, presented in Fig. 1, is a common workflow pattern [18]. The connector controls the execution of two components A and B such that either A executes, or B executes, or both execute. The connector then synchronises on the completion of whichever of A and/or B ran.

2.2 Connector colouring: an overview

The connector colouring (CC) semantics for Reo [8] is based on colouring the ends of a connector using 2 colours to represent the presence or absence of data flow. A more refined version using 3 colours enabled context-dependent semantics. In each state, each primitive has a set of possible *colourings* that 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*. These tables are composed by considering each entry in one table with every entry in the other and joining compatible ones. Compatibility

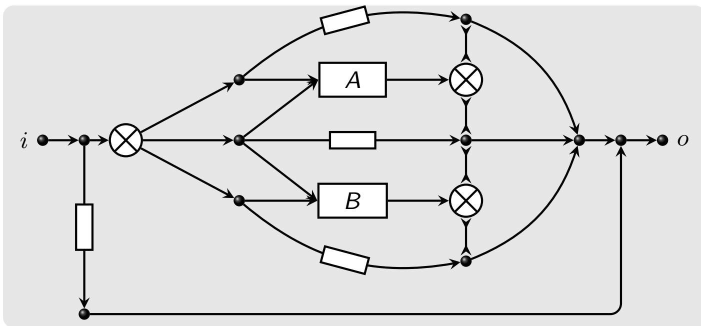

Fig. 1. Synchronous merge connector.

Primitives and their colouring tables Connector and its colouring table

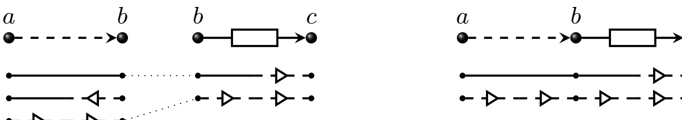

Fig. 2. Connector colouring. The dotted lines indicates compatible colourings.

between colourings is determined by ensuring that they match where two ends are joined to form a node. Fig. 2 illustrates the idea.

2-Colouring The 2-colouring scheme consists of colours ——•, which denotes the presence of data flow, and - - -•, which denotes its absence. For orientation, the • indicates the end. When plugging together colouring tables to compute the behaviour of connectors, both colours match only with themselves. That is, ——•—— and - - -•- - - are the only valid combinations.

3-Colouring The 3-colouring scheme is used to express context-dependent behaviour. A primitive depends on its context if its behaviour changes non-monotonically with increases in possibilities of data flowing on its ends [8, 6]. That is, by adding more possibilities of data flow, the primitive actually rules out already valid behaviour possibilities. One colour (——•) marks ends in the connector where data flows, and two colours mark the absence of data flow (-◁-• and -▷-•). The arrow indicates the *reason for no flow*, which can be either because no data is being written or because connector or component is not ready to receive the data. The arrow flows from the cause. -◁-• denotes that the reason for no-flow originates from the context, and we say that the end *requires a reason* for no flow. Dually, -▷-• indicates that the reason for no-flow originates from the primitive and we say that the end *gives a reason* for no-flow. Two 3-colours

Channel	Representation	Colouring table
LossySync	a - - - - - → b	a •———• b a •———◁-• b a •-▷--▷-• b
Priority Merger	a, b → c	a, b, c
FIFOEmpty$_1$	a —▭→ b	a •———▷-• b a •-▷--▷-• b

Table 1. Colouring tables for some primitives.

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: ——•——, -◁-•-◁-, -▷-•-◁- and -▷-•-▷-, but not -◁-•-▷-. Table 1 presents the colouring tables for some primitives. Fig. 2 gives a small example illustrating how 3-colouring works.

2.3 Notation

Let $\mathcal{X}$ be the global set of node names. Let $\mathcal{X}^{\updownarrow} = \{x^{\downarrow} \mid x \in \mathcal{X}\} \cup \{x^{\uparrow} \mid x \in \mathcal{X}\}$. Here $x^{\downarrow}$ denotes that x is a source end, and $x^{\uparrow}$ denotes that x is a sink end. Let '$\circ$' range over the set $\{\uparrow, \downarrow\}$; define $\overline{\circ}$ as $\overline{\uparrow} = \downarrow$ and $\overline{\downarrow} = \uparrow$. Let $\mathcal{P}$ be the set of primitives in a connector. Let P range over subsets of $\mathcal{P}$. Function $ends : \mathcal{P} \to \mathbb{P}(\mathcal{X}^{\updownarrow})$ gives the ends of a primitive. $nodes : \mathcal{P} \to \mathbb{P}(\mathcal{X})$ gives the set of nodes (which is the set of ends with the direction marking removed). These functions can be lifted to operate on $\mathbb{P}(\mathcal{P})$ in the obvious way. Function $internal : \mathbb{P}(\mathcal{P}) \to \mathbb{P}(\mathcal{X})$ gives all the internal nodes and $boundary : \mathbb{P}(\mathcal{P}) \to \mathbb{P}(\mathcal{X}^{\updownarrow})$ gives all the boundary ends for connector with primitives P. These are constrained by $internal(P) \cap boundary(P)^{\ddagger} = \emptyset$, where $(-)^{\ddagger}$ removes the $\uparrow$ or $\downarrow$. A collection of primitives $\mathcal{P}$ is a *well-formed connector* whenever $\forall p, p' \in \mathcal{P} \cdot ends(p) \cap ends(p') = \emptyset$. That is each node in $\mathcal{P}$ appears at most once as a sink and once as a source.

2.4 Formalism

Connector colouring is formalised as follows, adapting the original description [8]. The basis of connector colouring is a *colouring scheme*, a tuple $(\mathcal{C}olour, \frown)$, consisting of a set of colours and a symmetric *match* relation $\frown \subseteq \mathcal{C}olour \times \mathcal{C}olour$ that determines when two colours may be plugged together.

Definition 1 (Connector Colouring). *Given colouring scheme* $(\mathcal{C}olour, \frown)$. *A* colouring *for connector* P *is a function* $c : X \to \mathcal{C}olour$, *where* $X = ends(P) \subseteq \mathcal{X}^{\updownarrow}$, *that maps each end to a colour such that for all* $x \in internal(P)$, $c(x^{\uparrow}) \frown c(x^{\downarrow})$. *A* colouring table *over ends* $X \subseteq \mathcal{X}^{\updownarrow}$ *is a set of colourings with domain* X. *Two*

colouring tables over disjoint sets of ends are compatible, denoted $c_1 \frown c_2$*, whenever* $\forall x \in \mathcal{X} \cdot x^\circ \in dom(c_1) \wedge x^{\overline{\circ}} \in dom(c_2) \Rightarrow c_1(x^\circ) \frown c_2(x^{\overline{\circ}})$*. The* product *of two colouring tables* T_1 *and* T_2 *with disjoint domains, denoted* $T_1 \odot T_2$*, is the following colouring table, whose domain is the union of the domains of* T_1 *and* T_2*,* $T_1 \odot T_2 \ = \ \{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, c_1 \frown c_2\}$.

2-colouring is given by colouring scheme where $\mathcal{C}olour = \{$——•, - - -•$\}$, where binary relation $\frown$ is '='. 3-colouring is given by $\mathcal{C}olour = \{$——•, -▷-•, -◁-•$\}$ and $\frown = \{\langle$——•, ——•$\rangle, \langle$-◁-•, -▷-•$\rangle, \langle$-▷-•, -◁-•$\rangle, \langle$-▷-•, -▷-•$\rangle\}$.

The second colouring from Fig. 2, namely a •——— ◁- • b, is formalised as colouring $\{a^\downarrow \mapsto$ ——•, $b^\uparrow \mapsto$ -◁-•$\}$. It models the scenario where end a has data flow and end b has no data flow but requires a reason for its absence.

A stateful connector with ends X and states Q can be modelled using automata with transition structure $\Delta \subseteq Q \times (X \to \mathcal{C}olour) \times Q$, such that state changes occur only when flow occurs at some end.

3 Problem Statement

The problem with connector colouring semantics (and the encoding into constraints based on them) is that it takes a global view of the connector, that is, all primitives of the connector are considered at every round. This has two negative consequences. Firstly, it eliminates the opportunity to exploit the concurrent, potentially asynchronous, evolution of distinct parts of a connector. Secondly, it means that implementations based on connector colouring will not be scalable.

Consider the connector in the top left-hand corner of Fig. 3. This consists of a FIFO_1 connected to priority merger. On the boundary are three components: Both A and B are attempting to write a single message, while C is able to accept as many messages as possible. The different boxes in the figure correspond to different states of this connector. The transitions between these states are labelled with a 3-colouring if one exists, and a 2-colouring if not. (Valid 3-colourings can be converted to valid 2-colourings by forgetting the direction-of-reason arrows.)

Ideally, all transitions except the dotted one should be permitted—the dotted transition violates the preference of the priority merger. However, 2-colouring allows all transitions, which is too lax. On the other end of the spectrum, the only reachable transitions permitted by 3-colouring are those indicated with a thick line. 3-colouring rules out the dotted transition, but it also rules out other possibilities. Consider, for example, the transitions from the start state. These correspond to allowing writers A and B to succeed independently or together. But 3-colouring allows only both succeeding together. The other two behaviours are reasonable, as there is no channel synchronising actions A and B.

4 Partial Connector Colouring

The problems described in the previous section are solved by partial connector colouring. The idea is simple: only part of the connector is coloured to determine

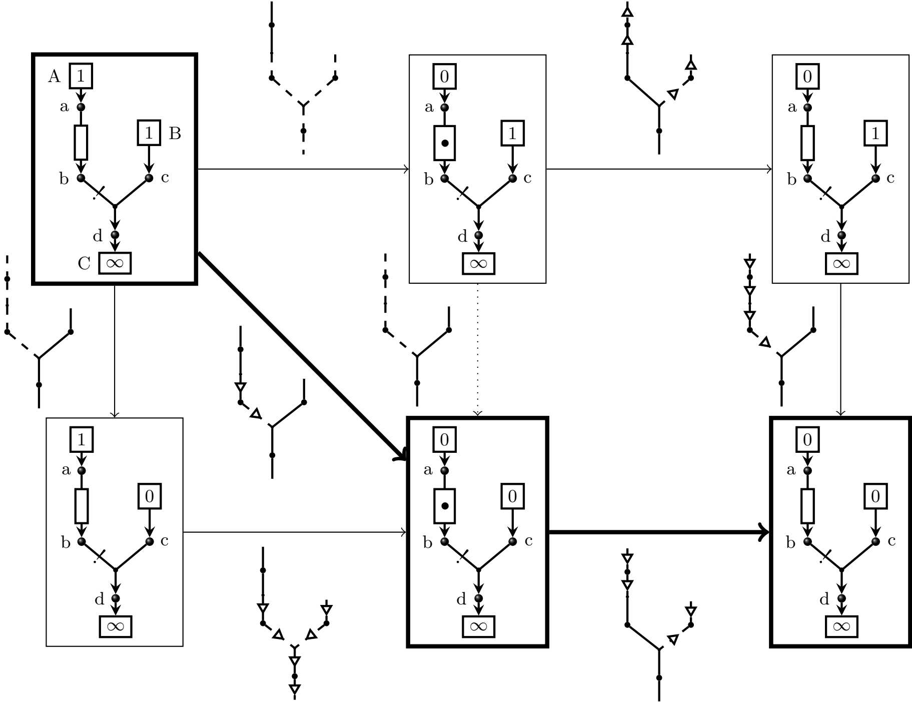

Fig. 3. Evolution of example connector.

the behaviour. The part that is not coloured plays no role in the computation. The challenge is determining which partial colourings are sensible; this comes down determining where the boundary of the coloured part can be draw and what that boundary may look like.

Partial connector colouring is based on partial colouring schemes, which are triples $(\mathcal{C}olour, \frown, \mathsf{valid})$, where $\mathcal{C}olour$ is a set of colours, $\frown \subseteq (\mathcal{C}olour + 1) \times (\mathcal{C}olour + 1)$ is a symmetric matching relation such that $\bot \frown \bot$, and $\mathsf{valid} : \mathcal{C}olour \to 2$ is a predicate that states which colours can appear on the boundary of the coloured part, defined as $\mathsf{valid}(x)$ if and only if $x \frown \bot$.

Definition 2 (Partial Connector Colouring). *Given a partial colouring scheme* $(\mathcal{C}olour, \frown, \mathsf{valid})$*. A* partial colouring $c : X \rightharpoonup \mathcal{C}olour$ *for* $X \subseteq \mathcal{X}^{\updownarrow}$ *is a partial function from ends to colours. The remainder of the definition (partial colouring tables and table composition) follows Definition 1, with partial colourings instead of total colourings.*

A light gray colour will be used to denotes places where no colouring is computed—that is where the colouring table is undefined. Valid partial colourings satisfy two well-formedness conditions on the internal ends and boundary ends.

Definition 3 (Valid Partial Colouring). *A partial colouring* $c : X \rightharpoonup \mathcal{C}olour$ *for connector* P*, where* $X = ends(P)$*, is* valid *iff for all* $x \in internal(P)$*,* $c(x^{\uparrow}) \frown c(x^{\downarrow})$ *and* $\mathsf{valid}(x^{\circ})$ *holds for all* $x^{\circ} \in boundary(P)$*.*

Partial 2-colouring is defined as $\mathcal{C}olour = \{$——•, - - -•$\}$ and $\frown = \{\langle$——•, ——•$\rangle$, $\langle$- - -•, - - -•$\rangle$, $\langle$- - -•, $\bot\rangle$, $\langle\bot$, - - -•$\rangle$, $\langle\bot, \bot\rangle\}$. Only valid(- - -•) holds. A valid partial 2-colouring is one where no data flows at the boundaries.

Partial 3-colouring as $\mathcal{C}olour = \{$——•, -▷-•, -◁-•$\}$ and $\frown = \{\langle$——•, ——•$\rangle$, $\langle$-◁-•, -▷-•$\rangle$, $\langle$-▷-•, -◁-•$\rangle$, $\langle$-▷-•, -▷-•$\rangle\langle$-▷-•, $\bot\rangle$, $\langle\bot$, -▷-•$\rangle$, $\langle\bot, \bot\rangle\}$ Only valid(-▷-•) holds. A valid partial 3-colouring is one where all boundary ends give a reason for no flow.

Both partial 2- and 3-colouring schemes are well-behaved with respect to composition; the composition of two valid colourings produces a valid colouring.

The level of granularity expressed thus far is that of whole primitives, but we can go finer. For some primitives, the behaviour of some ends is independent of its other ends. This means that different parts of the primitive can be coloured (or not coloured) independently. This kind of behaviour is called *split colouring*.

We illustrate split colourings using the FIFO_1 buffer. Consider the partial colouring of the FIFO_1 buffer is it its empty state in Fig. 4(a). It is the case that both ends of this primitive are independent (in the empty state), so the colour tables could be presented as in Fig. 4(b), where the FIFO_1 is actually treated as two separate primitives. Based on this, the effective colouring table for the FIFO_1 buffer is the one in Fig. 4(c).

The remainder of the figure revisits our example connector. Fig. 4(d) shows an attempt to find a colouring without split colourings for one of the transitions leaving the initial state, but this colouring is not valid due to the node marked ×. This corresponds to a writer blocking for no reason. With split colourings, the colouring in Fig. 4(e) can instead be used. This colouring does not even look at the writer; it is not blocked, just not involved in the computation.

Two conditions govern the validity of a split colouring. The first is that when we split the colouring table for the connector and then rejoin it, as in Fig. 4(a–c), no additional data flow behaviour is introduced. The second condition deals with stateful primitives. As each split colouring is derived from some partial

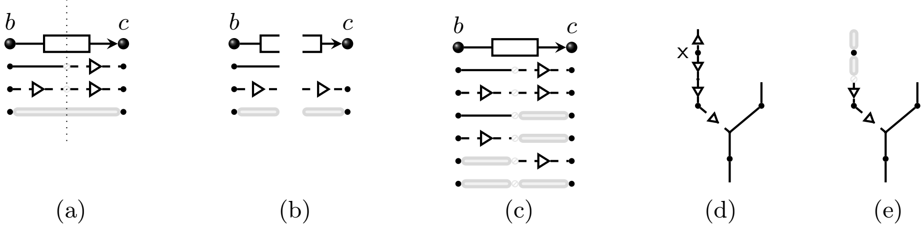

Fig. 4. Splitting the colouring table of an empty FIFO_1 buffer (a–c). Candidate connector without split colourings (d) and with them (e)

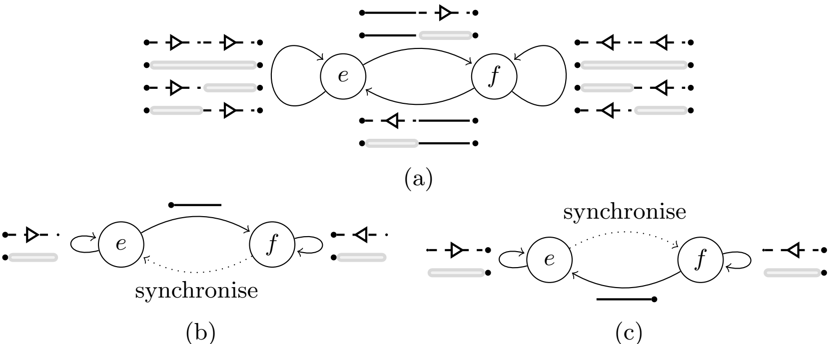

Fig. 5. Splitting the automaton of a FIFO_1. The automata for the parts of the split FIFO_1 have an *out-of-phase* synchronisation step to keep the same view on the state.

colouring, which itself may cause a change in the state, consistency demands that no no-flow colouring can result in a state change *and* that any combination of split colourings do not result in different state changes—there is at most one state change among the split colourings.

When only one part causes a change of state, the two parts need to synchronise to ensure that both have the same view on the state. We illustrate with an example. Fig. 5(a) gives an automaton for a FIFO_1, where transitions are labelled with the split colourings that cause the respective transitions—each transition has multiple colourings. When the primitive is split it results in the two automata shown in Fig. 5(b–c), one for each end of the FIFO_1. Now, if a data flow transition is taken in the first automaton in state e, the second automaton is not aware of the state change, as it has no behaviour to change the state. In order for them to get into the same state, an *out-of-phase* synchronisation step is made between the two ends of the split primitive. Formal conditions capturing the correctness criteria have been explored in Proença's thesis [16].

Fig. 6 presents what partial 3-colouring will allow. Split colouring of the FIFO_1 primitive is required for the transitions marked **1** and **2**. Not only does partial 3-colouring with split colouring allows all desirable behaviours, but it also reduces the computation required to achieve these, thereby increasing scalability. It allows *mosaic evolution*, whereby different parts of the connector can evolve at different rates, increasing concurrency. Partial and split colourings thus offer more potential concurrency, as the ends can be computed independently, better scalability, as the size of the connector being considered can be smaller, and more behavioural possibilities.

5 Constraint-based Encoding

In previous work we showed how to encode connector colouring as constraints [9]. The constraints encode three aspects of behaviour: *synchronisation constraints*

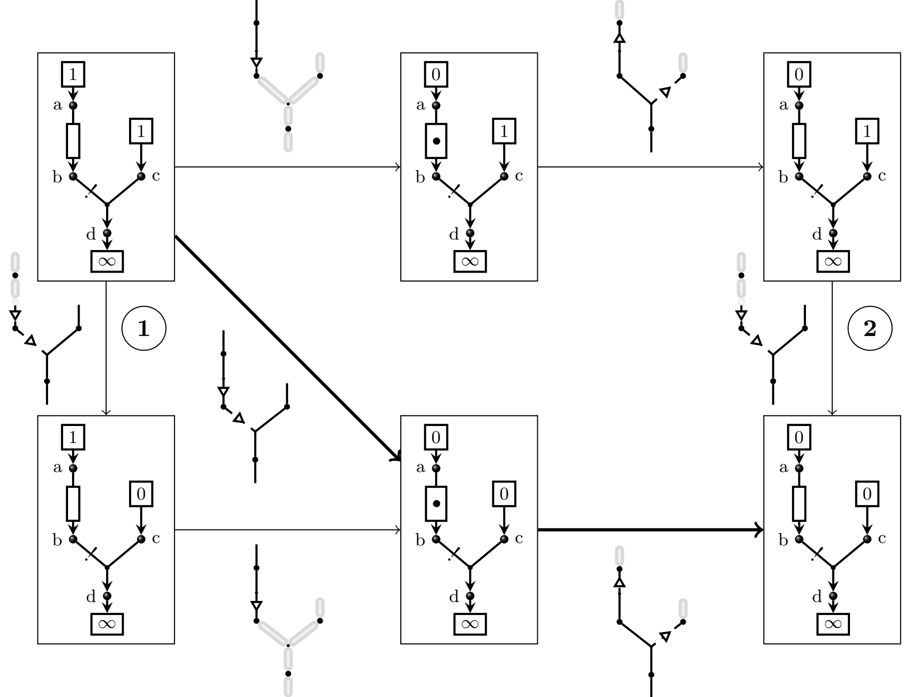

Fig. 6. Evolution of example connector with partial colouring.

(SC) describe the presence or absence of data flow at each end—that is, whether or not those ends synchronise; *data flow constraints* (DFC) describe the data flow at the ends that synchronise (which connector colouring does not capture); and *context constraints* (CC) describe the direction of the reasons for no-flow. For encoding 2-colouring, context constraints can be dropped. State and state transitions can also be encoded, but we omit them here for brevity.

Constraints are defined over (1) propositional synchronisation variables $\mathcal{X}$, (2) data flow variables $\widehat{\mathcal{X}} = \{\widehat{x} \mid x \in \mathcal{X}\}$ ranging over $\mathcal{D}ata_{\perp} \mathrel{\widehat{=}} \mathcal{D}ata \cup \{\texttt{NO-FLOW}\}$, where $\mathcal{D}ata$ be the domain of data and where $\texttt{NO-FLOW} \notin \mathcal{D}ata$ represents 'no data flow', and (3) propositional context variables $\mathcal{X}^{\updownarrow}$. The encoding of colourings into constraints is given in the following table:

Colouring	Constraint	Colouring	Constraint
$x^{\downarrow} \mapsto$ ——•	x	$x^{\uparrow} \mapsto$ ——•	x
$x^{\downarrow} \mapsto$ -▷-•	$\neg x \wedge x^{\downarrow}$	$x^{\uparrow} \mapsto$ -▷-•	$\neg x \wedge x^{\uparrow}$
$x^{\downarrow} \mapsto$ -◁-•	$\neg x \wedge \neg x^{\downarrow}$	$x^{\uparrow} \mapsto$ -◁-•	$\neg x \wedge \neg x^{\uparrow}$

Constraints are expressed as quantifier-free, first-order logical formulas. Table 2 presents the semantics of some commonly used primitives. A *solution* to a formula ψ is an assignment of variables, defined in the usual manner [9].

Channel	Representation	SC	DFC	CC
Sync	$a \longrightarrow b$	$a \leftrightarrow b$	$\widehat{a} = \widehat{b}$	$a \vee \neg a^{\downarrow} \vee \neg b^{\uparrow}$
LossySync	a - - - - - → b	$b \rightarrow a$	$b \rightarrow (\widehat{a} = \widehat{b})$	$\neg a \rightarrow (\neg b \wedge \neg a^{\downarrow} \wedge b^{\uparrow}) \ \wedge$ $\neg b \rightarrow ((a \wedge \neg b^{\uparrow}) \vee \neg a)$
Priority Merger	a, b → c	$(c \leftrightarrow (a \vee b))$ $\wedge \neg(a \wedge b)$	$a \rightarrow (\widehat{c} = \widehat{a}) \wedge$ $b \rightarrow (\widehat{c} = \widehat{b})$	$(c \wedge \neg a) \rightarrow \neg a^{\downarrow} \ \wedge$ $(c \wedge \neg b) \rightarrow b^{\downarrow} \ \wedge$ $\neg c \rightarrow ((\neg a^{\downarrow} \wedge \neg b^{\downarrow}) \vee \neg c^{\uparrow})$
Replicator	a → b, c	$(a \leftrightarrow b) \wedge$ $(a \leftrightarrow c)$	$\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a}$	$a \vee \neg a^{\downarrow} \vee \neg b^{\uparrow} \vee \neg c^{\uparrow}$
FIFO_1 (empty)	a –[ ]→ b	$\neg b$	$\top$	$(\neg a \rightarrow \neg a^{\downarrow}) \wedge b^{\uparrow}$
FIFO_1 (full)	a –[d]→ b	$\neg a$	$b \rightarrow (\widehat{b} = d)$	$a^{\downarrow} \wedge (\neg b \rightarrow \neg b^{\uparrow})$

Table 2. Channel Encodings. SC=synchronisation constraints. DC=data flow constraints. CC=context constraints.

The following axiom captures the relationship between the different kinds of variables, and it is applicable to all ends in a connector:

$$(\neg x \leftrightarrow \widehat{x} = \texttt{NO-FLOW}) \wedge (\neg x \rightarrow x^{\uparrow} \vee x^{\downarrow}) \qquad \text{(flow axiom)}$$

The first clause captures that `NO-FLOW` is used as the value of a data flow variable when no flow occurs, which is the same as when the corresponding synchronisation variable is $\bot$. The constraint $x^{\uparrow} \vee x^{\downarrow}$ is interpreted as saying that the reason for no data flow either comes from the sink end, from the source end, or from both ends. The second clause is dropped when using 2-colouring. Let $\texttt{Flow}(X)$ denote the conjunction $\bigwedge_{x \in X}(\neg x \leftrightarrow \widehat{x} = \texttt{NO-FLOW}) \wedge (\neg x \rightarrow x^{\uparrow} \vee x^{\downarrow})$.

The constraints for an entire connector are obtained by taking the conjunction of the constraints of the primitives making up the connector and adding in the flow axiom for the ends present. For example, the connector in Fig. 2 is encoded in constraints as follows:

$$\begin{aligned} \Psi_{SC} &= b \rightarrow a \wedge \neg c \qquad \Psi_{DFC} \ = \ b \rightarrow (\widehat{a} = \widehat{b}) \wedge \top \\ \Psi_{CC} &= \neg a \rightarrow (\neg b \wedge \neg a^{\downarrow} \wedge b^{\uparrow}) \wedge \neg b \rightarrow ((a \wedge \neg b^{\uparrow}) \vee \neg a) \wedge (\neg b \rightarrow \neg b^{\downarrow}) \wedge c^{\uparrow} \\ \Psi &= \Psi_{SC} \wedge \Psi_{DFC} \wedge \Psi_{CC} \wedge \texttt{Flow}(\{a, b, c\}). \end{aligned}$$

5.1 Partial 2- and 3-Constraints

Now we can modify the constraints generated to handle partial colouring. The first difference is that the constraints will be applied not to all primitives,

but to a subset of the primitives—split primitives treated as separate primitives. First, let $constraints_p$ denote the constraints for connector p, and define $constraints_P = \bigwedge_{p \in P} constraints_p$. Two kinds of additional are constraints generated. The first ensures that the solution to the constraints corresponds to a valid partial colouring by constraining the boundary:

$$\bigwedge_{x^\circ \in boundary(P)} \neg x \wedge x^\circ \qquad \text{(boundary)}$$

The 2-colouring variant is obtained by dropping the '$\wedge x^\circ$' part. It is straightforward to show that the collection of constraints faithfully encodes partial 2/3-colouring. An additional kind of constraint is added to check whether there is any flow within a connector, as ultimately we are only interested in partial colourings where something happens:

$$\bigvee_{x \in internal(P)} x \qquad \text{(internal-flow)}$$

6 Implementation and Benchmarks

Partial connector colouring is implemented using constraint satisfaction in a fashion similar to our constraint-based implementation of Reo [9]. The previous implementation collected all the constraints for a connector and solved them. The resulting solution described the behaviour in the connector.

To exploit partiality, a different approach is required. This involves trying to solve the constraints for a part of the total connector; if no solution is found, a larger part is tried. More specifically, each thread operating on a connector performs the following steps. The thread is initially given a number of primitives, which it is said to *own*. It collects the constraints for those primitives, including the boundary and internal flow constraints, and solves the resulting constraint satisfaction problem. A solution corresponds to flow within that part of the connector, so the thread ensures that the corresponding data is moved to where it belongs and the states of stateful primitives change. If, however, there is no solution, then the part of the connector the thread is operating on is expanded and the algorithm repeats. When two threads attempt to *own* the same part of the connector, one claims both parts of the connector and the other thread is returned to the thread pool to be reassigned work.

Due to the nature of this approach, we cannot expect that it will always be better—implementations will need to be tuned to specific connectors. Nevertheless, a number of preliminary benchmarks do show that partial connector colouring can produced a faster implementation. The benchmarks consist of two parametric connectors, depicted in Fig. 7. These connector each have a part that can be repeated a given number of times. We ran four versions of our implementation on these connectors on a 8-core 2.4 GHz Intel Xeon desktop with 16 GB RAM running Ubuntu Linux. As a baseline, the first version implemented the original connector colouring model. The other versions implement partial connector colouring, using one, two and four threads. These results show that partial colouring can lead to a faster, more concurrent, and scalable implementation.

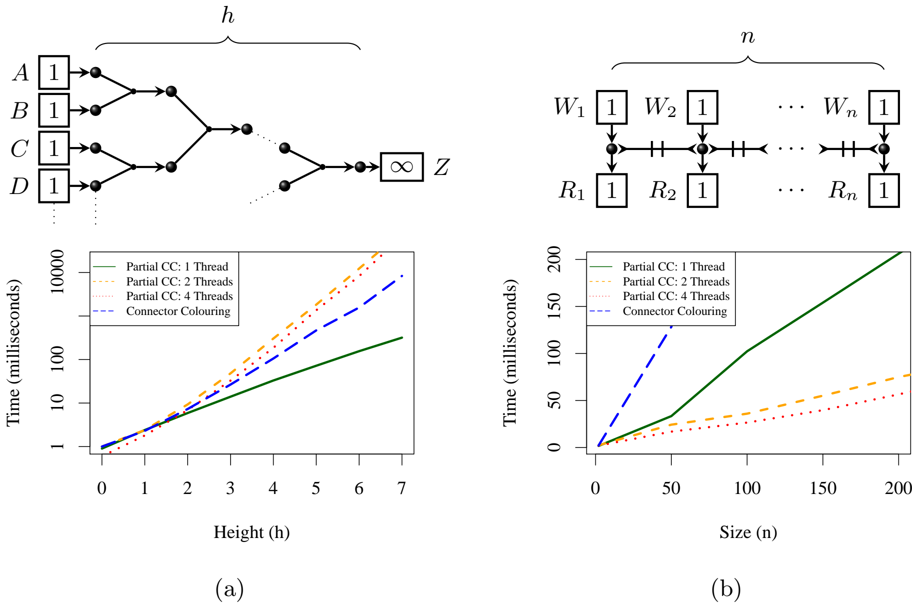

Fig. 7. Benchmarks: Multiple Merger (a) demonstrates that partial connector colouring offers an order of magnitude improvement for one thread, though multiple threads create too much contention. Pairwise Asynchronous (b) demonstrates significant advantages of partial colouring, which scales well as more threads (cores) are added.

There are however a number of dimensions where heuristics can be applied to improve the performance of the implementation, such as choosing the number of threads, choosing where they start computing, and choosing how the set of primitives considered grows. These will be investigated in future work.

7 Related work

Many semantic models exist for Reo. Clarke et al. [6] outline the difference between various models. Only connector colouring [8], Reo automata [6], intentional automata [10], and the tile model [4] capture context dependent behaviour. Connector colouring was the first to do so, and the others refine it in one way or another. The models are all distinct—and the relationship between them has not been investigated—but for our purposes, we do not consider automata-based models as these do not lead to scalable implementations. The constraint-based encoding of Reo [9] offers orders of magnitude improvement in performance and scalability of Reo implementations.

Wegner describes coordination as constrained interaction [19] and Montanari and Rossi express coordination as a constraint satisfaction problem [15].

They describe how to solve synchronisation problems using constraint solving techniques, but they do not encode context dependency or identify the benefits of partiality. Lazovik et al. [13] utilise constraints to provide a choreography framework for web services. The choreography, the business processes, and the requests are modelled as a set of constraints. Their setting, however, cannot express the same kinds of coordination patterns as Reo. The analogy between Reo and constraint satisfaction has already been made [3], and indeed Klüppelholz and Baier encode the constraint automata model for Reo in terms of binary decision diagrams in order to model check Reo connectors [12]. 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 our constraint encoding is phrased in terms of a richer data domain.

Minsky and Ungureanu introduce the Law-Governed Interaction (LGI) mechanism [14], implemented by the Moses toolkit, where in laws are constraints specified in a Prolog-like language, enforced on regulated events of the agents, such as sending or receiving messages. This mechanism targets distributed coordination of heterogenous agents using a policy that enforces extensible laws. However, laws are local, in the sense that can only refer to the agent being regulated. While this allows LGI to achieve good performance, it limits expressiveness.

Bruni et al.'s stateless algebra of connectors [7] and Sobociński's [17] stateful extension resemble the core of Reo, though they lack context dependency and data. The semantics is similar to 2-colouring in this limited setting. Bliduze and Sifakis [5] present a semantics for their BIP coordination model in terms of boolean propositions. Although similar, BIP is not as expressive as Reo. Jongmans et al. [11] present an encoding of 3-colouring in terms of 2-colouring that resembles our encoding into propositional constraints. Their approach adds fictitious nodes into the Reo level, whereas ours is completely under the surface.

8 Conclusion

This paper presented partial connector colouring which improves on the existing connector colouring semantics for Reo and their subsequent encoding as constraints in three respects. Firstly, the semantics enable a more scalable implementation by avoiding the lock-step evaluation of an entire connector. Secondly, the amount of concurrency is increased by enabling different parts of a connector to evolve independently at different rates. Finally, partial 3-colouring retains the context dependent behaviour of 3-colouring but regains some of the behavioural possibilities lost due to the lock-step computation. Benchmarks have also been presented to confirm the possible performance improvements.

An interesting direction for future work is a more in depth empirical exploration of the heuristics alluded to in Section 6.

References

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

2. F. Arbab, C. Koehler, Z. Maraikar, Y. Moon, and J. Proença. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In *International Workshop on Formal Aspects of Component Software (FACS)*, Malaga, 2008. Electronic Notes in Theoretical Computer Science (ENTCS).
3. Farhad Arbab. Composition of interacting computations. In D. Goldin, S. Smolka, and P. Wegner, editors, *Interactive Computation: The New Paradigm*, pages 277–321, Secaucus, NJ, USA, 2006. Springer-Verlag New York, Inc.
4. Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese, and Ugo Montanari. Tiles for Reo. In *Proceedings of Recent Trends in Algebraic Development Techniques (WADT)*, volume 5486 of *LNCS*, pages 37–55, Berlin, Heidelberg, 2008. Springer.
5. Simon Bliudze and Joseph Sifakis. Synthesizing glue operators from glue constraints for the construction of component-based systems. In *Software Composition*, volume 6708 of *Lecture Notes in Computer Science*, pages 51–67, 2011.
6. Marcello M. Bonsangue, Dave Clarke, and Alexandra Silva. Automata for context-dependent connectors. volume 5521 of *LNCS*, pages 184–203, 2009.
7. Roberto Bruni, Ivan Lanese, and Ugo Montanari. A basic algebra of stateless connectors. *Theor. Comput. Sci.*, 366(1-2):98–120, 2006.
8. D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, May 2007.
9. D. Clarke, J. Proença, A. Lazovik, and F. Arbab. Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.*, 76(8):681–710, 2011.
10. David Costa. *Formal Models for Component Connectors*. PhD thesis, Vrij Universiteit Amsterdam, 2010.
11. Sung-Shik T. Q. Jongmans, Christian Krause, and Farhad Arbab. Encoding context-sensitivity in reo into non-context-sensitive semantic models. In *Coordination Models and Languages*, volume 6721 of *LNCS*, pages 31–48, 2011.
12. Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *ENTCS*, 175(2):19–37, 2007.
13. A. Lazovik, M. Aiello, and R. Gennari. Choreographies: Using constraints to satisfy service requests. In *Proc. of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, page 150, Washington, DC, USA, 2006. IEEE Computer Society.
14. Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, 2000.
15. Ugo Montanari and Francesca Rossi. Modeling process coordination via tiles, graphs, and constraints. In *3rd Biennial World Conference on Integrated Design and Process Technology*, volume 4, pages 1–8, 1998.
16. José Proença. *Synchronous coordination of distributed components*. PhD thesis, LIACS, Leiden University, May 2011.
17. P. Sobociński. Representations of Petri net interactions. In *Concurrency Theory (CONCUR '10)*, number 6269 in LNCS, pages 554–568. Springer, 2010.
18. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
19. P. Wegner. Coordination as constrained interaction (extended abstract). In *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 28–33, 1996.