Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 229 (2009) 43–58
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Deconstructing $\mathcal{R}$eo

Dave Clarke[2] José Proença[2] Alexander Lazovik[1,3]
Farhad Arbab[2]

CWI, Amsterdam, The Netherlands

Abstract

Coordination in $\mathcal{R}$eo emerges from the composition of the behavioural constraints of the primitives, such as channels, in a component connector. Understanding and implementing $\mathcal{R}$eo, however, has been challenging due to interaction of the *channel metaphor*, which is an inherently local notion, and the non-local nature of constraint propagation imposed by composition. In this paper, the channel metaphor takes a back seat, and we focus on the behavioural constraints imposed by the composition of primitives, and phrase the semantics of $\mathcal{R}$eo as a constraint satisfaction problem. Not only does this provide a clear *intensional* description of the behaviour of $\mathcal{R}$eo connectors in terms of *synchronisation* and *data flow constraints*, it also paves the way for new implementation techniques based on constraint propagation and satisfaction. In fact, decomposing $\mathcal{R}$eo into constraints provides a new computational model for connectors, which we extend to model interaction with an unknown external world beyond what is currently possible in $\mathcal{R}$eo.

Keywords: Coordination, Constraint Satisfaction, Constraint Automata, $\mathcal{R}$eo, Connector Colouring

de•con•struct verb [trans.]
analyze (a text or linguistic or conceptual system) by deconstruction, typically in order to expose its hidden assumptions and contradictions and subvert its significance or unity.

1 Introduction

Wegner describes coordination as constrained interaction [16]. We take this approach literally and represent coordination using constraints. Specifically, we take the view that a $\mathcal{R}$eo [5] 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 not only makes it easier to understand $\mathcal{R}$eo connectors, we claim, but it also opens the door to more efficient implementation techniques—a claim supported by preliminary experimental results—and to alternative ways of thinking about 'channel-based' coordination.

[1] Work carried out during an ERCIM "Alain Bensoussan" Fellowship.
[2] Email: {`dave`,`proenca`,`farhad`@cwi.nl}
[3] Email: `lazovik@gmail.com`

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.06.028

$\mathcal{R}$eo is generally presented as a channel-based coordination language wherein component connectors are compositionally constructed out of *primitives*, which are typically 2-ended channels.[4] The behaviour of connectors are described in terms of the constraints imposed by the channels and their composition, in terms of three of kinds of constraints: (1) data is accepted on an input channel end if by accepting it the channel can satisfy its behavioural constraints; (2) data is offered by an output channel end if by offering the data the channel can satisfy its behavioural constraints; and (3) *nodes* connecting the channel ends (1:1, in the direction of data flow) must pass on any data they receive, that is, data offered by one channel end must be accepted by the other. This is all achieved under the restriction that the only communication between entities occurs though the channels.

When it comes to implementing $\mathcal{R}$eo, a number of *challenges* arise.

Challenge 1: Strained Metaphor As a specification model, $\mathcal{R}$eo can be equated with electrical circuits or water flow through pipes. However, such systems have a natural, equilibrium-based realisation, whereas the metaphor does not extend to the implementation of $\mathcal{R}$eo. *It is impossible to make choices that are local to channels in order to satisfy the constraints imposed by the entire connector.* Some form of backtracking or global arbitration is required.

Constraint automata [8] and connector colouring [9] provide the basis for both semantic models and implementations, as well as for model checking and visualisation tools for $\mathcal{R}$eo. *Constraint automata* provide the semantics of each primitive and composition in $\mathcal{R}$eo, by representing the synchronisation possible in a connector, along with a description of the data flow, in an automata-based model. The actual constraints are based on the state of the primitives, and transitions in the automaton correspond to data flow in the connector. *Connector colouring* is based on the simple idea that ends where data flows and data does not flow in a connector can be coloured with different colours. Each primitive has a set of possible *colourings* describing its possible behaviours. The semantics of a connector is determined by plugging together the colourings of the primitives in such a way that the *colours match*, meaning that data flow into a node is the same as the flow out of the node.

Challenge 2: Implementation Impositions No natural model of $\mathcal{R}$eo exists. All implementations directly implement some semantic model, so limitations in the semantic model are inherited by implementations based in that model.

Constraint automata provide such a comprehensive description of behaviour that channels actually become redundant. This is the case with virtually all connectors presented in the literature.[5] The connector colouring based implementation—which is the only one suitable as the basis for a distributed implementation—can only describe the synchronisation constraints of connectors, but cannot handle data-aware behaviour, such as filters.

[4] We use the words *channel* and *primitive* interchangeable, though prefer the latter as it lifts the somewhat arbitrary limitation to 2-ended channels.

[5] It is also the case that constraint automata-based implementations are inherently centralised, and in fact lose all potential parallelism, because a connector is implemented as a single automaton, but we do not address this problem in this paper.

Restricting communication so that it can only occur through channels, as in our distributed implementation [2], prohibiting a global agent or direct node-to-node communication, imposes additional obligations on the channels. They are required to play a significant role in the global constraint resolution process, such as passing around colouring tables and being the conduits for all communication.

Challenge 3: Limited support for external primitives Research on $\mathcal{R}$eo focusses exclusively on the connectors, without much consideration on the interaction with the *unknown* outside world. Indeed, constraint automata models preclude any external primitives. Primitives of interest may include data transformers or filters whose details are externally computed.

In this paper, we address these challenges by adopting a different view of $\mathcal{R}$eo. The channel/circuit-view of a $\mathcal{R}$eo connector becomes a mere metaphor. Instead, a $\mathcal{R}$eo 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 constraint-based approach to $\mathcal{R}$eo is developed in three 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 as a constraint; and

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.

The resulting model significantly extends $\mathcal{R}$eo, with both data-aware and externally defined behaviour, enabling more efficient implementation techniques.

This paper is organised as follows. We elaborate on $\mathcal{R}$eo in Section 2 using an example. Section 3 describes our encoding of $\mathcal{R}$eo-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. Section 5 presents the main contribution of the paper, namely, a reformulation of $\mathcal{R}$eo as iterative and interactive constraint satisfaction. Section 6 and 7 present related work and our conclusions.

2 $\mathcal{R}$eo Coordination Model

$\mathcal{R}$eo [5,6] is a channel-based coordination model, wherein coordinating *connectors* are constructed compositionally out of more primitive connectors. Primitives can include mergers and replicators and channels offering a variety of behavioural policies regarding synchronisation, buffering, lossiness, and even direction of data flow. Communication with a primitive occurs through its ports, called *ends*: data enters the primitive through a *source end*, and data is produced upon *sink ends.*[6] 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. For the purposes of this paper, we do not distinguish between primitives such as channels used for coordination and the components being coordinated. Typically, the 'coordinator' has more control over the choice of the behaviour of primitives, whereas component behaviour is externally determined.

Rather than reiterate previous accounts of $\mathcal{R}$eo's semantics, we present a description directly in terms of constraints. The first thing to note is that the behaviour of each primitive depends upon its current state. The semantics are described *per-state* in a series of *rounds*. *Data flow* on an end occurs when a single datum is passed through that end. Within a particular round data flow may occur on some number of ends. This is equated with the notion of *synchrony*.

The semantics are 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 permits data flow either on both of their ends or on neither end, and asynchronous channel types permits 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 flowing on the source end of a synchronous channel is the same as the data flowing on its sink end; or that there is no constraint on the data flow, such as for a drain which simply discards its data; or it might say that the data satisfies a particular predicate, in the case of a filter channel.

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.*[7] Data flows through a connector from primitive to primitive through nodes, observing the policy that nodes cannot buffer data. This means that the two ends in a node are synchronised and have the same data flow—behaviourally, they are equal.

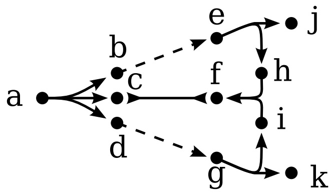


Fig. 1. Exclusive router connector.

The following example illustrates $\mathcal{R}$eo's semantics. The connector in Fig. 1 is an exclusive router built by composing two LossySync channels (b-e and d-g),

[6] This naming corresponds to sources and sinks in directed graphs. Sometimes *input* and *output* are used instead of source and sink.

[7] Other descriptions of $\mathcal{R}$eo use more general $n : m$ nodes, but it has been shown [8,9] that 1:1 (sink-to-source, that is, output-to-input) plugging plus a notion of merger and replicator is equivalent.

one SyncDrain (c-f), one Merger (h-i-f), and three Replicators (a-b-c-d, e-j-h and g-i-k). The informal semantics of each of these primitives is as follows:

LossySync *b-e* data flow at end b is always possible. If data flows at b, data flow at e is also possible, in which case the data that flows at ends b and e is equal.

SyncDrain *c-f* data flows at end c if and only if it flows at end f, though there is no constraint on the value of the data.

Replicator *e-j-h* data flows at either all ends or no end, and the value of the data at ends j and h is the value at the end e. The '3-replicator' a-b-c-d behaves similarly, and the value of the data at ends b, c, and d is the value at the end a.

Merger *h-i-f* data flows on ends h and f (and not on end i) or on ends i and f (and not on end h), where the data is equal on both ends where data flows.

In all cases, it is possible that there is no data flow at all. The semantics of the nodes are transparently handled by using the same name for its two ends.

These constraints can be combined to give the following two behavioural possibilities (plus the no flow anywhere possibility):

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

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

The next section gives a formal definition of the primitives used in this example, from which we can verify that their composition yields the expected behaviour.

3 Coordination via Constraint Satisfaction

This section formalises the per-round semantics of $\mathcal{R}$eo primitives and composition as a set of constraints. The possible coordination patterns are then determined using traditional constraint satisfaction techniques.

3.1 Mathematical Preliminaries

Let $\mathcal{X}$ be the set of ends in a connector. $\widehat{\mathcal{X}}$ denote the variables of set $\mathcal{X}$ with little hats. Let $\mathcal{D}ata$ be the domain of data, and define $\mathcal{D}ata_\perp \mathrel{\widehat{=}} \mathcal{D}ata \cup \{\texttt{NO-FLOW}\}$, where $\texttt{NO-FLOW} \notin \mathcal{D}ata$ represents 'no data flow.' Constraints are expressed in quantifier-free, first-order logic over two kinds of variables: *synchronisation variables* x, which are propositional (boolean) variables, and *data flow variables* $\widehat{x}$, which are variables over $\mathcal{D}ata_\perp$, where $x \in \mathcal{X}$. Constraints are formulæ in the following grammar:

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

where $d \in \mathcal{D}ata_\perp$ 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: $\perp \mathrel{\widehat{=}} \neg\top$; $\psi_1 \vee \psi_2 \mathrel{\widehat{=}} \neg(\neg\psi_1 \wedge \neg\psi_2)$; $\psi_1 \rightarrow \psi_2 \mathrel{\widehat{=}} \neg\psi_1 \vee \psi_2$; and $\psi_1 \leftrightarrow \psi_2 \mathrel{\widehat{=}} (\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1)$.

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

$$
\begin{array}{ll}
\sigma, \delta \models \top \text{ always} & \sigma, \delta \models \psi_1 \wedge \psi_2 \ \textit{iff} \ \sigma, \delta \models \psi_1 \text{ and } \sigma, \delta \models \psi_2 \\
\sigma, \delta \models x \ \textit{iff} \ \sigma(x) = \top & \sigma, \delta \models \neg\psi \qquad \textit{iff} \ \sigma, \delta \not\models \psi \\
\multicolumn{2}{l}{\sigma, \delta \models P(t_1, \ldots, t_n) \ \textit{iff} \ (\mathit{Val}_\delta(t_1), \ldots, \mathit{Val}_\delta(t_n)) \in \mathcal{I}(P)}
\end{array}
$$

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

3.2 Frame Axiom

The following constraint captures the relationship between no flow on end $x \in \mathcal{X}$ and the value `NO-FLOW`:

$$\neg x \leftrightarrow \widehat{x} = \texttt{NO-FLOW} \qquad \text{(frame axiom)}$$

Let $\texttt{Frame}(\mathcal{X})$ denote $\bigwedge_{x \in \mathcal{X}} (\neg x \leftrightarrow \widehat{x} = \texttt{NO-FLOW})$. A solution that satisfies $\texttt{Frame}(\mathcal{X})$ is called a *firing*. As we are only interested in finding firings, so we assume that the frame 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 in each end—that is, whether those ends synchronise. The latter constraints involve also data flow variables from $\widehat{\mathcal{X}}$ to describe the data flow on the ends that synchronise.

Fig. 2 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 has two states indicating that it is empty (FIFOEmpty$_1$) and full with data d (FIFOFull$(d)_1$). Curiously, the constraints for some channels, such as SyncDrain and SyncSpout, are identical. In $\mathcal{R}$eo, the directions of the data flow is used to govern the well-formedness of connector composition, but our constraints ignore it.

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

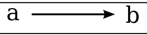
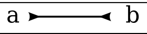
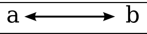
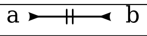
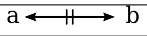
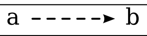
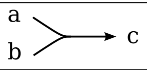
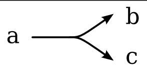
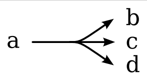
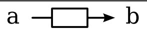
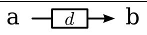
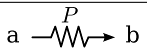
Channel	Representation	SC	DFC
Sync		$a \leftrightarrow b$	$\widehat{a} = \widehat{b}$
SyncDrain		$a \leftrightarrow b$	$\top$
SyncSpout		$a \leftrightarrow b$	$\top$
AsyncDrain		$\neg(a \wedge b)$	$\top$
AsyncSpout		$\neg(a \wedge b)$	$\top$
LossySync		$b \rightarrow a$	$b \rightarrow (\widehat{a} = \widehat{b})$
Merger		$(c \leftrightarrow (a \vee b)) \wedge \neg(a \wedge b)$	$a \rightarrow (\widehat{c} = \widehat{a}) \wedge b \rightarrow (\widehat{c} = \widehat{b})$
Replicator		$(a \leftrightarrow b) \wedge (a \leftrightarrow c)$	$\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a}$
3-Replicator		$(a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (a \leftrightarrow d)$	$\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a} \wedge \widehat{d} = \widehat{a}$
FIFOEmpty$_1$		$\neg b$	$\top$
FIFOFull$(d)_1$		$\neg a$	$b \rightarrow (\widehat{b} = d)$
Filter(P)		$b \rightarrow a$	$b \rightarrow (P(\widehat{a}) \wedge \widehat{a} = \widehat{b}) \wedge (a \wedge P(\widehat{a})) \rightarrow b$

Fig. 2. Channel Encodings

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

LossySync channel A LossySync can always accept 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.

FIFOEmpty$_1$ and FIFOFull$(d)_1$ FIFO$_1$ 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 Mergers permit data flow synchronous through one of its source ends, exclusively, to its sink end.

Replicator Replicators (and 3-replicators) allow data to flow only synchronously at every channel end. Data is replicated from the source end to every sink end.

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

Other channels could use non-trivial predicates of more than one argument. An example is a special synchronous drain variant whose predicate $P(\widehat{a}, \widehat{b})$ requires that the location element the data on $\widehat{a}$ is nearby the location of $\widehat{b}$.

Splitting the constraints into synchronisation and data flow constraints is very natural, and it closely resembles the constraint automata model [8]. It also enables some implementation optimisations, if we require that the synchronisation constraints are an abstraction of the overall constraints.[8] Following Sheini and

[8] Given overall constraints ψ and synchronisation constraints ψ_{SC}, require that $\sigma, \delta \models \psi \;\Rightarrow\; \sigma \models \psi_{SC}$.

Sakallah [14], 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 guarantees a solution to the data flow constraints. The only primitive in Fig. 2 for which this is not true is the filter, as it inspects the data in order to determine the 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. If the constraints for the two connectors are ψ_1 and ψ_2, then the constraints for their composition is simply $\psi_1 \wedge \psi_2$.

Top-level constraints are given by the following grammar:

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

where ψ is as before. Existential quantification is present to abstract away intermediate channel ends. The satisfaction relation is extended with the rules:

$$\begin{array}{lll} \sigma, \delta \models \exists x.\mathcal{C} & \textit{iff} & \text{there exists a } b \in \{\top, \bot\} \text{ such that } \sigma, \delta \models \mathcal{C}[b/x] \\ \sigma, \delta \models \exists \widehat{x}.\mathcal{C} & \textit{iff} & \text{there exists a } d \in \mathcal{D}ata_\bot \text{ such that } \sigma, \delta \models \mathcal{C}[d/\widehat{x}]. \end{array}$$

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

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

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

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

$$\sigma_1 \;=\; a \wedge j \wedge \neg k \qquad \sigma_2 \;=\; a \wedge \neg j \wedge k \qquad \sigma_3 \;=\; \neg a \wedge \neg j \wedge \neg k$$

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

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

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

4 Reconstructing $\mathcal{R}$eo: Adding State

4.1 Encoding State Machines

Primitives such as the FIFO$_1$ channel are stateful, i.e., their state and subsequent behaviour changes after data flows through the channel. This is exemplified in the constraint automata (CA) semantics of $\mathcal{R}$eo [8]. The CA of a FIFO$_1$ channel is shown in Fig. 3. Its initial state is q_0. From this state it can take a transition to state $q_1(d)$ if there is data flow on end a, excluding data flow on end b. The constraint $d = \widehat{a}$ holds corresponds to storing the value flowing on end a in the internal state d. The transition from $q_1(d)$ to q_0 is read in a similar way, except that the data is moved from the internal state of $q_1(d)$ to end b.

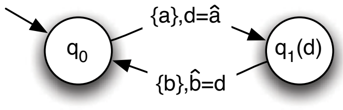


Fig. 3. Constraint Automata for the FIFO$_1$ channel

To encode state information, our logic is extended so that terms also include n-ary uninterpreted function symbols (data is a 0-ary uninterpreted function symbol):

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

A term t is *ground* iff $t = f(t_1, \ldots, t_n)$ and each t_i for $1 \leq i \leq n$ is ground.

Let I be the set of stateful primitives in a connector. Add a new set of term variables $state_p$ and $state'_p$, where $p \in I$, to denote the state before and after the present step. State machines of primitives are encoded in the standard manner [11].[9] For example, the state machine of a FIFO$_1$ channel is encoded as the formula:

$$state = \texttt{empty} \rightarrow \begin{cases} \neg b \ \wedge \\ a \rightarrow state = \texttt{full}(\widehat{a}) \ \wedge \\ \neg a \rightarrow state' = state \end{cases} \wedge \quad state = \texttt{full}(d) \rightarrow \begin{cases} \neg a \ \wedge \\ b \rightarrow \widehat{b} = d \wedge state' = \texttt{empty} \ \wedge \\ \neg b \rightarrow state' = state \end{cases}$$

To complete the encoding, a formula describing the present state is added to the mix. In the example, the formula $state = \texttt{empty}$ records the fact that the FIFO$_1$ is in the empty state, and $state = \texttt{full}(d)$ 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 $\bigwedge_{p \in I} state_p = t_p$. This is called a *(pre-)state vector*. Similarly, $\bigwedge_{p \in I} state'_p = t_p$, 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 need to contribute to the state vector.

[9] Both the correctness and compositionality of our encoding wrt constraint automata are straightforward.

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

Constraint satisfaction techniques can now form the working heart of the $\mathcal{R}$eo *engine*. The engine holds the current set of constraints (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 some constraints to capture the new state. This is depicted in the diagram in Fig. 4.

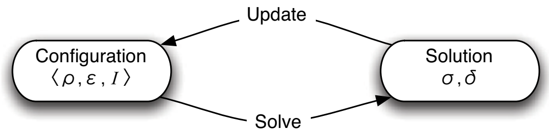


Fig. 4. Phases of the $\mathcal{R}$eo Engine

The *configuration* of the engine is a triple $\langle \rho, \varepsilon, I\rangle$, where ρ represents *persistent* constraints, ε represents *ephemeral* constraints, and I is the set of stateful primitives in the connector. The former constraints are eternally true for a connector, such as the description of the state machines of the primitives. The latter includes 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:

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

satisfying the following:

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

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

The assumption made thus far is that all state information for a primitive is known in advance. In this case, there is no need to actually supply an implementation of the primitives, as they are redundant. This is already the case with the constraint automata-based implementation of $\mathcal{R}$eo [3].

The next section describes how to encode primitives and components whose state is not known. We call the primitives described in this section *internal primitives*.

5 Reconstructing $\mathcal{R}$eo: Adding Externals

This section describes *external primitives* whose entire behaviour is not *a priori* available to the constraint solver, and the extensions to the engine required to support them. Two types of external primitives are presented: those with external state and those using external functions or predicates.

5.1 External State

External stateful primitives provide ephemeral constraints for only a single coordination round. In each update phase, these constraints are replaced by a new set of constraints which are determined by interacting with the external primitive. That is, the following communication between the $\mathcal{R}\mathsf{eo}$ engine and primitive p takes place:

$$\begin{tikzcd} \mathit{engine} \arrow[r, "\mathit{state}'_p = \mathtt{s}^n"] & \boxed{p} \arrow[r, "\psi_p^{n+1}"] & \mathit{engine} \end{tikzcd}$$

where $\mathtt{s}^n$ is a term describing the state selected by the $\mathcal{R}\mathsf{eo}$ engine at the end of the solve stage, and ψ_p^{n+1} is the next round of constraints for primitive p. The constraints are defined over the primitive's ends and state'_p. This means that the variable state'_p will take on a value at the end of the *solve* step. The state constraint is used to pass information including data and the selected behaviour to the primitive. For example, a constraint such as $\mathit{state}'_p = \mathtt{res}(\widehat{a}, \widehat{b})$ encodes that the data on ends a and b are passed to the primitive p. In addition, the primitive may use the uninterpreted function symbol `res` to encode information to guide its external activity. Note that this encoding and way of interacting with primitives means that *data is not passed through the channel ends, but to the primitive directly.*

A configuration will now be a quadruple $\langle \rho, \varepsilon, I, E \rangle$, where ρ, ε, and I are as before, and E is the set of external primitives used in the connector. The ephemeral constraints for $n+1$ in the presence of external primitives are defined as follows:

$$\varepsilon^{n+1} \equiv \bigwedge_{p \in I} \mathit{state}_p = \delta^n(\mathit{state}'_p) \wedge \bigwedge_{p \in E} \psi_p^{n+1} \qquad \text{(update-updated)}$$

To illustrate the interaction between the $\mathcal{R}\mathsf{eo}$ engine and external primitives, consider the example of a component giving the current temperature every time unit, connected by a Sync channel to a component that receives and displays data. The connector is depicted in Fig. 5.

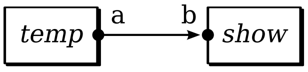


Fig. 5. $\mathcal{R}\mathsf{eo}$ connector of a thermometer connected to a display.

The boxes represent external primitives, called *temp* and *show*. Each has a single end—*temp* produces data on end a, and *show* accepts data on end b. A possible configuration of the entire connector is $\langle \rho, \varepsilon, \emptyset, \{\mathit{temp}, \mathit{show}\} \rangle$, where

$$\begin{aligned} \rho &\equiv (a \leftrightarrow b) \wedge (\widehat{a} = \widehat{b}) \wedge \mathtt{Frame}(a, b) \\ \varepsilon &\equiv (a \rightarrow \widehat{a} = \mathsf{20^\circ C}) \wedge (\mathit{state}'_{\mathit{temp}} = *) \wedge \\ &\quad\; (b \rightarrow \mathit{state}'_{\mathit{show}} = \mathtt{print}(\widehat{b})) \wedge (\neg b \rightarrow \mathit{state}'_{\mathit{show}} = *) \end{aligned}$$

The persistent constraint ρ describes the behaviour of the Sync channel and the frame axioms for its ends. The ephemeral constraint ε expresses the behaviour for the current round. The first part gives constraints for *temp*, and the second part gives constraints for *show*. The dummy value, $*$, is used to denote the state in cases where no flow occurs. This is required as each external primitive's constraints will be updated in the update round.

Other possible behaviours for the thermometer and display include:

- A thermometer offering $0°C$ or $32°F$ has synchronisation constraint $\top$ and data flow constraint $a \to (\widehat{a} = \texttt{0°C} \vee \widehat{a} = \texttt{32°F})$, describing a choice of possible data values. Note that components in the existing $\mathcal{R}$eo model cannot express this sort of behaviour.
- Display only data satisfying ≥ 0 has synchronisation constraint $\top$ and data flow constraints $a \to \widehat{a} \geq 0$

In both cases, if the primitives are not offering/accepting any data, the synchronisation constraint would be $\neg a/\neg b$, and the data flow constraint would be $\top$, which states that there is no additional constraint.

5.2 External functions and predicates

We extend our approach by incorporating *external predicates* and *external function symbols* to model predicates and functions which are not (or cannot be) represented directly as constraints. These enable more fine-grained interaction with external primitives during the *solve* phase.

Syntactically, constraints and terms are extended, respectively, with predicates and function symbols indexed by external primitives, such as $P_p(t_1, \ldots, t_n)$ and $f_p(t_1, \ldots, t_n)$, where $p \in E$. During constraint satisfaction, the $\mathcal{R}$eo engine interacts with the corresponding primitive to 'evaluate' the external symbols, as follows:

$$\begin{tikzcd} engine \arrow[r, "f_p(t_1,\ldots,t_n)?"] & \boxed{p} \arrow[r, "t_{res}"] & engine \end{tikzcd}$$

Here the engine asks the primitive to evaluate the function f_p with arguments $t_1, \ldots, t_n$, and the primitive returns result t_{res}. External predicates are evaluated in a similar fashion. For consistency, we assume that primitive p returns the same answer to each call of f_p or P_p with the same arguments. This interaction with the primitive does not cause the primitive to change state, as this is achieved using the mechanism described in the previous section.

To more formally describe how this extension integrates with the solve phase, we present an abstract description of the constraint satisfaction process and adapt it to include interaction with external primitives.

The constraint satisfaction process [4] can be described abstractly as a relation $\longrightarrow \subseteq \textsc{Constraints} \times \textsc{Constraints}$, satisfying at least the following condition:

$$C \longrightarrow C' \quad \textit{if} \quad C' \text{ implies } C$$

Read $C \longrightarrow C'$ as C *reduces to* C'. The idea is that $\longrightarrow$ captures possible branching and simplifications that occur during constraint satisfaction, reducing, ultimately, to a terminal constraint. A constraint is *terminal* if it is either $\bot$ or a conjunction of literals (propositional variables or their negation) and equalities of the form $\widehat{x} = g$, where g is a ground term. Let $\longrightarrow^*$ be the transitive closure of $\longrightarrow$.

We extend this relation with additional possible choices to model external interaction. Whenever the engine sees a constraint of the form $C \wedge t = f_p(t_1, \ldots, t_n)$ or

$C \wedge P_p(t_1, \ldots, t_n)$ it can interact with the primitive p evaluate the function or constraint. This is modelled by adding two *labelled* transitions to the relation, where t_i and t_{res} are ground, and $b_{res} \in \{\top, \bot\}$:

$$C \wedge t = f_p(t_1, \ldots, t_n) \xrightarrow{f_p(t_1,\ldots,t_n)=t_{res}} C \wedge t = t_{res}$$
$$C \wedge P_p(t_1, \ldots, t_n) \xrightarrow{P_p(t_1,\ldots,t_n)=b_{res}} C \wedge b_{res}$$

Worked Example

The thermometer example presented in Fig. 5 is now adapted so that the primitive *temp* obtains the current temperature, *during the solve state*, via an external 0-argument function $\texttt{current}_{temp}$, and the primitive *show* uses a predicate $\texttt{Acceptable}_{show}$ to determine when its argument is acceptable to display. The new ephemeral and overall constraints are:

$$\begin{aligned}\varepsilon \equiv &(a \rightarrow \widehat{a} = \texttt{current}_{temp}) \wedge (state'_{temp} = *) \wedge \\ &(b \rightarrow \texttt{Acceptable}_{show}(\widehat{b})) \wedge (b \rightarrow state'_{show} = \texttt{print}(\widehat{b})) \wedge (\neg b \rightarrow state'_{show} = *)\\ \varphi \equiv &(a \leftrightarrow b) \wedge (\widehat{a} = \widehat{b}) \wedge \texttt{Frame}(a, b) \wedge \varepsilon.\end{aligned}$$

The following illustrates a number of internal and external steps which may be taken during the constraint satisfaction process (for appropriate ψ and ϕ):

$$\begin{aligned}\varphi &\xrightarrow{\qquad\qquad\qquad}{}^{*} \psi \wedge \widehat{a} = \texttt{current}_{temp}\\ &\xrightarrow{\texttt{current}_{temp}=20^\circ\texttt{C}} \psi \wedge \widehat{a} = 20^\circ\texttt{C}\\ &\xrightarrow{\qquad\qquad\qquad}{}^{*} \phi \wedge \widehat{b} = 20^\circ\texttt{C} \wedge \texttt{Acceptable}_{show}(20^\circ\texttt{C})\\ &\xrightarrow{\texttt{Acceptable}_{show}(20^\circ\texttt{C})=\top} \phi \wedge \widehat{b} = 20^\circ\texttt{C} \wedge \top\\ &\xrightarrow{\qquad\qquad\qquad}{}^{*} a \wedge b \wedge \widehat{a} = 20^\circ\texttt{C} \wedge \widehat{b} = 20^\circ\texttt{C} \wedge\\ &\qquad state'_{temp} = * \wedge state'_{show} = \texttt{print}(20^\circ\texttt{C})\end{aligned}$$

5.3 Component interaction

Interaction between components and the engine in our model differs from previous descriptions of $\mathcal{R}$eo, as depicted in Fig. 6.

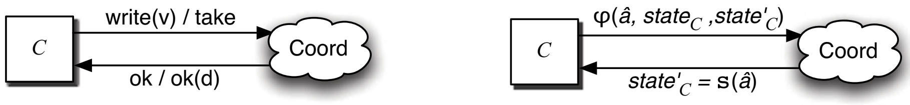


Fig. 6. Interaction with primitives from $\mathcal{R}$eo (left) and our perspective (right).

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 some subsequent round (including the present one) the engine replies, with a possible data value.

In our model, components play a more participative role, wherein they publish a 'meta-level' description of their possible behaviour in the current round. The engine replies with a term which the component interprets as designating its new state and, if required, the data flow that occurred.

5.4 Some implementation issues

Currently, we have implemented an exploratory prototype $\mathcal{R}$eo engine incorporating most of the features discussed in this paper, based on the constraint solver Choco [1]. Compared to existing $\mathcal{R}$eo implementations, performance results are encouraging.

The implementation addresses a number of issues. Firstly, it is well known that the *variable ordering* has a huge impact on the performance of constraint solvers [4]. The topology of a connector can be used to help determine the variable ordering. For example, it is better to start from a source of data, such as an external component or a full FIFO_1 buffer. Secondly, to ensure fairness in the implementation—that is, avoiding that the same choice is repeatedly taken—requires some randomisation in the variable ordering. Finally, to avoid, where possible, the solution corresponding to no data flow in the connector, whenever the search process branches on a synchronisation variable, the *true*-branch, corresponding to flow, is explored first.

6 Related work

Wegner describes coordination as constrained interaction [16], yet surprisingly little work takes this literally, representing coordination as constraints. Montanari and Rossi express coordination as a constraint satisfaction problem, in a similar but more general way [12]. They describe how to solve synchronisation problems using constraint solving and propagation. 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 in these terms, giving a clear meaning for each variable, and describing the interaction with the external world within the solve and update stages.

The interaction with external entities is similar to the model explored by Faltings *et al* [10] in a constraint satisfaction setting. They introduce a framework of open constraint satisfaction in a distributed environment which allows new constraints to be added on-the-fly. The set of variables is fixed, but the data domain can be extended by querying a third party for the next value. Their approach also considers weighted constraints to find optimal solutions. This may be adapted to our setting for various notions of priority. The main difference between our work and theirs is that we focus on the coordination of third parties, making a cleaner distinction between computation and coordination. In addition, the extra topology information provided by connectors can be used to improve the search algorithms.

Klüppelholz and Baier describe a symbolic model checking approach for $\mathcal{R}$eo [11]. 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. Their model does not consider external interaction to the degree ours does.

The timed concurrent constraint (tcc) programming framework [15] was introduced by Saraswat *et al.* to integrate the concurrent constraint (cc) programming paradigm [13] 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 a constraint 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.

Coordination models have been applied to coordinate solvers of distributed constraint satisfaction problems (DCSP) [7]. Ironically, our coordination model is based on (D)CSP.

7 Conclusion and Future Work

Let's return to the original challenges to see how the reconstruction of $\mathcal{R}$eo fares.

Challenge 1: Strained Metaphor Our model abandoned channels as an implementation concept. Instead, meta-level interaction (with external entities) is required to resolve the global constraints imposed by channels and their composition. By abandoning channels, the implementation is free to optimise (persistent) constraints, eliminating costly infrastructure. Channels remain as a specification-level metaphor.

Challenge 2: Implementation Impositions Primitives in our model must report at least their next step behaviour. They need to commit to any proposed behaviour, and answer consistently whenever re-asked. Observable actions are performed at the end of a phase. This is quite an imposition, though no more than existing implementations. Our model *does not* require that channels also implement part of the $\mathcal{R}$eo engine, which is the case with our distributed engine.

Challenge 3: External Primitives The model uses external states, functions and predicates to capture the interaction with and coordination of arbitrary external entities. This makes our model more flexible than constraint automata and connector colouring as a basis for implementing $\mathcal{R}$eo.

The constraint-based approach offers the possibility of using existing research and tools to develop an efficient implementation of $\mathcal{R}$eo. Constraints also provide a flexible framework, so it may be possible in the future to mix-in other constraint based notions, such as service-level agreements. Future work will explore these directions, in particular, the increased expressiveness. We will also try to exploit the parallelism inherent in constraints.

References

[1] *Choco constraint programming system*, http://choco.sourceforge.net/.

[2] *Distributed coordination with Reo*, http://www.cwi.nl/themes/sen3/distributedreo.

[3] *Eclipse coordination tools*, http://homepages.cwi.nl/~koehler/ect.

[4] Apt, K. R., "Principles of Constraint Programming," Cambridge University Press, 2003.

[5] Arbab, F., *Reo: a channel-based coordination model for component composition*, Math. Struct. in Comp. Science **14** (2004), pp. 329–366.

[6] Arbab, F., *Abstract behavior types: a foundation model for components and their composition*, Science of Computer Programming **55** (2005), pp. 3–52.

[7] Arbab, F. and E. Monfroy, *Coordination of heterogeneous distributed cooperative constraint solving*, SIGAPP Appl. Comput. Rev. **6** (1998), pp. 4–17.

[8] Baier, C., M. Sirjani, F. Arbab and J. Rutten, *Modeling component connectors in Reo by constraint automata*, Sci. Comput. Program. **61** (2006), pp. 75–113.

[9] Clarke, D., D. Costa and F. Arbab, *Connector colouring I: Synchronisation and context dependency*, Science of Computer Programming **66** (2007), pp. 205–225.

[10] Faltings, B. and S. Macho-Gonzalez, *Open constraint programming.*, Artif. Intell. **161** (2005), pp. 181–208.

[11] Klueppelholz, S. and C. Baier, *Symbolic model checking for channel-based component connectors*, in: *FOCLASA'06*, 2006.

[12] Montanari, U. and F. Rossi, *Modeling process coordination via tiles, graphs, and constraints*, in: *IDPT'98*, 1998.

[13] Saraswat, V. A., "Concurrent constraint programming," MIT Press, Cambridge, MA, USA, 1993.

[14] Sheini, H. M. and K. A. Sakallah, *From propositional satisfiability to satisfiability modulo theories*, in: A. Biere and C. P. Gomes, editors, *SAT*, Lecture Notes in Computer Science **4121** (2006), pp. 1–9.

[15] V.A.Saraswat, R.Jagadeesan and V.Gupta, *Timed default concurrent constraint programming*, Journal of Symbolic Computation **22** (1996), pp. 475–520, extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[16] Wegner, P., *Coordination as constrained interaction (extended abstract)*, in: *Coordination Languages and Models*, Lecture Notes in Computer Sciences **1061**, 1996, pp. 28–33.