Fundamenta Informaticae XX (2007) 1–30
IOS Press

A Basic Logic for Reasoning about Connector Reconfiguration

Dave Clarke
CWI
Amsterdam, The Netherlands
dave@cwi.nl

Abstract. Software systems evolve over time. From a component-based software engineering perspective, this means that either the components of the system need to change, or, if components are connected using a coordination layer, then the way in which they are connected needs to change, or both. In some situations, changes must be performed without stopping the running system. This not only introduces a serious technological challenge, it also makes reasoning about the evolving system difficult. One approach to this problem is to use component connectors to plug components together. Reconfiguration of a system can then be reduced to reconfiguring the component connector, as changing component implementations can be implemented by changing which components the connector connects together. The coordination language Reo offers operations to dynamically reconfigure the topology of component connectors, but until now, no means for reasoning about reconfiguration in Reo has been developed. This issue is addressed in this paper. To enable reasoning about connector behaviour, and hence behaviour of the composed system, we present a semantics of Reo in the presence of reconfiguration, and a logic together with its model checking algorithm.[1]

Keywords: Dynamic Reconfiguration, Coordination, Component Connectors, Reo.

1. Introduction

Software systems evolve over time. From a component-based software engineering perspective, this means that the components of a system need to be replaced by new versions which have had bugs fixed or new features added. Sometimes such updates require changes to the interface of a component, and this generally necessitates that other components in the system also change. In some situations, changes must be performed without stopping the running system. Continuously running distributed systems, in particular, require extensive support to facilitate evolution, deployment, upgrading, and reconfiguration.

[1]This paper expands an FSEN conference paper by the same author [18], adding more discussion and proofs of theorems.

Such evolution not only introduces a serious technological challenge, it also makes reasoning about the evolving system difficult.

Coordination languages [35] offer a technology to address this issue. Rather than just plugging components directly together, a coordination language provides a layer for connecting software components. The layer acts as a mediator which pumps data between the components, but it also offers the benefit of reducing the dependency of components on each other. Using a coordination layer offers immediate benefits in the presence of software evolution, as it can shift the focus of software evolution away from components in the following ways. Firstly, some changes to the system can be localised to the coordination layer. Secondly, changing a component can be seen, in part, as an operation on the coordination layer — the glue between components is changed to exclude the original component and incorporate the new component. Lastly, the presence of a coordination layer can also buffer the bulk of the components in the system from the changes required to one component, as, for example, the coordination layer can adapt the interface of the replacement component to conform with the behaviour expected by the remainder of the system. In the context of this paper, we deal with a coordination layer consisting of circuit-like connectors. In this setting, changing the coordination layer is synonymous with reconfiguring the connector. Hence to address the issue of software evolution, we concentrate on connector reconfiguration.

We focus on the channel-based coordination language Reo [2]. Reo provides circuit-like *connectors* for connecting software components in such a way that the components are unaware of their role in the composed software. Components and connectors are distinct entities. From our perspective, *components* are black boxes which operate by writing data to or reading data from known ports (which we call "ends"). Software components are connected at the boundary of the connector. A *channel* is a point to point communication means between its two ends. Reo has a more liberal notion of channel than usual: channels may buffer or may not buffer; the ends can either accept or produce data, in any combination, thus a channel may have two input ends or two output ends; operations on the ends may be synchronised, or be mutually exclusive; and the channel may even modify values passed through it. Reo enables *connectors* to be composed from smaller connectors, by plugging two or more channel ends together to form *nodes*.

The context we are working in assumes that reconfiguration of a connector occurs outside of the connector, as described in Arbab's article [2]. This implies that reconfiguration can affect the behaviour of the connector, but not vice versa. This makes sense in that we typically do not know *a priori* how a system will be reconfigured in the future, so reconfiguration need not be built into a connector. Our perspective is that software evolution requires changing the behaviour of the connector that connects the components in the software system. Changing the behaviour of a connector amounts to reconfiguring the connector, that is, changing the way in which channels are plugged together. Reo offers operations to dynamically reconfigure the topology of component connectors. The semantics of reconfiguration is clear: behave like the initial connector, reconfigure, then behave like the new connector. Without the proper precautions, however, reconfiguring running software is error-prone. Data sent to a component may not be received by its intended recipient if reconfiguration is performed at the wrong time, or more generally, the interleaving of reconfiguration steps and dataflow may violate a component's expected protocol. By guaranteeing the atomicity of certain operations, Reo's architecture aims to avoid some of this danger, but it cannot cover all possibilities, such as protocol faults.

Reasoning about system evolution requires formal models and logics. Until now, however, no means for reasoning about reconfiguration in Reo has been developed. To this end, this paper makes the following *contributions*. To enable reasoning about connector behaviour, and hence behaviour of the composed

system, we present: a semantic model for Reo connectors in the presence of reconfiguration; a logic for reasoning about reconfiguration of operating connectors; and a model checking algorithm for the logic.

Organisation: After reviewing Reo and giving some reconfiguration scenarios in this section, Sections 2 and 3 formalise Reo connectors and their reconfiguration. Section 4 reviews Reo semantics as constraint automata. Sections 5 and 6 present ReCTL*, a logic for reasoning in the presence of reconfiguration, and its model checking algorithm. Section 7 revisits the reconfiguration scenarios, and Sections 8 and 9 discuss related work and conclude the paper.

1.1. Overview of Reo

In this section we present an overview of Reo's component connectors. For a full account of Reo, see Arbab's articles [2, 3]. Reo is a channel-based coordination language based on circuit-like *connectors* that coordinate software components. Reo imposes a notion of *exogenous* coordination to orchestrate the components that are interconnected in a composed system. This means that coordination occurs outside of the components, and components are unaware of their role in the composed software.

Components interact with a Reo connector using a simple interface. A component will have access to a number of input and output ports, which will be the ends of channels. The only way a component may interact with a connector is by issuing I/O operations (*write* and *take*) on these ends. A *write* or *take* will succeed whenever the connector either accepts the data of the *write* or produces data for the *take*. The ability to accept or delay these operations forms the basis of coordination.

Channels constitute the only primitive connectors in Reo, each of which is a point-to-point communication medium with two distinct ends. Reo uses a generalized notion of channel. In addition to the common channel types of synchronous and asynchronous, with bounded or unbounded buffers, and with FIFO and other ordering schemes, Reo allows an open-ended set of channels, each with its own, sometimes exotic, behaviour. For instance, channels in Reo need not have both an input end—*accepting input*—and an output end—*producing output*; a channel can instead have two input ends or two output ends.

Sync	*SyncDrain*	*SyncSpout*	*LossySync*
AsyncDrain	*AsyncSpout*	$FIFO_1$	$FIFO_1(x)$

Figure 1. Some basic channel types in Reo

Figure 1 shows some example channels. *Sync* denotes a synchronous channel. Data flows through this channel if and only if it is possible simultaneously accept data on one end and pass it out the other end. *SyncDrain* denotes a synchronous drain. Data flows into both ends of this channel only if it possible to simultaneously accept the data on both ends. *SyncSpout* denotes a synchronous spout. Data flows out of both ends of this channel only if it possible to simultaneously *take* the data from both ends. The values emitted from each end are chosen non-deterministically from the data domain, possibly constrained to satisfy some predicate which may be associated with each end of the *SyncSpout*. *LossySync* denotes a lossy synchronous channel. If a *take* is pending on the output end of this channel and a *write* is issued

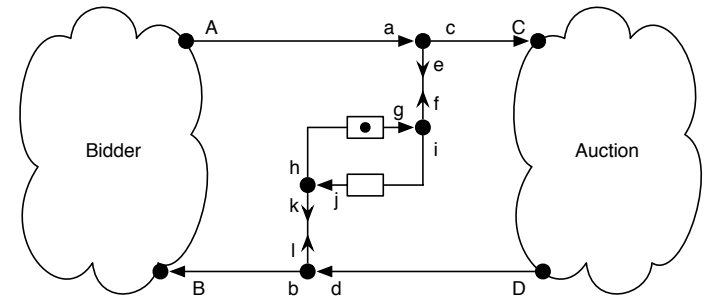
on the input end, then the channel behaves as a synchronous channel. However, if no *take* is pending, the *write* can succeed, but the data is lost. Note that this kind of lossy synchronous channel does not have a stochastic semantics, unlike other lossy channels appearing in the literature [38, 1]. *AsyncDrain* denotes an asynchronous drain. Data can flow into only one end of this channel at the exclusion of data flow at the other end. *AsyncSpout* denotes an asynchronous spout. Data can flow out of only one end of this channel at the exclusion of data flow at the other end. $FIFO_1$ denotes an empty FIFO buffer with capacity for one data item. Data can flow into the input end of this buffer, but no flow is possible at the output end. After data flows into the buffer, it becomes a full FIFO buffer. $FIFO_1(x)$ denotes a full FIFO buffer containing datum x. Data can flow out of the output end of this buffer, but no flow is possible at the input end. After data flows out of the buffer, it becomes an empty FIFO buffer.

More complex connectors can be constructed out of channels and simple connectors by conjoining channel ends to form *nodes*. A node may contain any number of channel ends. We classify nodes into three different types depending on the types of their coincident ends: an *input node* contains only input channel ends; an *output node* contains only output channel ends; and a *mixed node* contains both kinds of channel end. Nodes route data through a connector. A node may have any number of channel ends which push data into and accept data from it. Data flows at a node whenever both *exactly one* of the data suppliers (a component or an output end of a channel) can succeed in sending some data and *all acceptors* (a component or an input end of a channel) can *synchronously* accept that data. The synchronisation and exclusion constraints imposed by nodes and channels propagate through the entire connector. This leads to a powerful language of component connectors [2, 3].

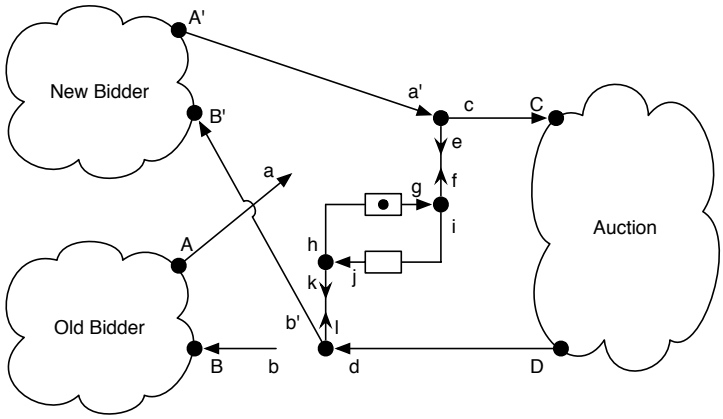
It is natural to consider Reo connectors in isolation. A connector can be viewed as an open system, in that, while it does not have any observable behaviour without plugging in components, its behaviour can be viewed as a relationship between the possible actions on its boundary nodes. An example connector is shown in Figure 2(a). The behaviour of the connector is to first allow A and C to succeed synchronously with data flowing from A to C, then allows B and D to succeed synchonously, with data flowing from D to B. Afterwards A and C may again succeed. The loop of two $FIFO_1$ buffers sequences these two events. The intuition is that when both A and C are ready, data flows synchronously from A, through node ace, and into C. Data flow at ace causes data to flow into the synchronous drain e-f. For this to occur, data must also flow at f. This data is supplied by the $FIFO_1(x)$ buffer, which pumps data into node gfi. This data is also copied into channel end i, thus filling the $FIFO_1$ i-j. As a result of this step, the buffer g-h becomes empty and i-j becomes full.

Although Reo connectors may look like electrical circuits and the synchronous channels may lead the reader to think of Reo connectors as synchronous systems (as in Esterel [12]), it would be wrong to equate Reo with either model. Although the precise implementation details are more involved, a Reo connector is executed in essentially two steps: (1) based on pending *write*/*take* operations, solve the synchronisation/exclusion constraints imposed by the channels of a connector to determine where data can flow; and (2) send data in accordance with the solution in step (1). The second step may not occur if no data flow is possible. In between step (2) and step (1) of the following round, new *write*/*take* operations may be attempted on the channel ends, or existing ones may be retracted due to time-out. Not all of the connector needs to be involved in step (1) at the same time: FIFO buffers, for example, serve to divide connectors into *synchronous slices* which operate more or less independently.

Reo is designed so that connectors can be deployed in a distributed setting, with components and Reo nodes assigned to various machines across the network. We do not require that channels are mapped in a way that follows the connections of the network, though our implementation assumes that channels



(a)



(b)

Figure 2. (a) Connector joining Bidder and Auction Components. The Bidder connects to nodes A (bid) and B (response) and the Auction connects to C and D. (b) Reconfiguration disconnects nodes A and B and adds connections to nodes A' and B'.

could be laid out in such a manner. This means that the underlying implementation must follow the topology of a connector when performing communications, which in principle enables Reo to be used in situations of limited connectivity, such as wireless sensor networks.

Reo also provides operations for constructing and reconfiguring connectors: operations for creating new channels, joining two nodes together, splitting a node in two, hiding internal nodes and forgetting boundary nodes. Before describing these operations in Section 3, we present a number of reconfiguration scenarios to motivate the reasoning apparatus presented in this paper.

1.2. Reconfiguration Scenarios

Auction Consider a distributed system with two kinds of components: one managing an auction and one issuing an individual bidder's bids. Bids are routed via a Reo connector, see Figure 2(a), to an auction component, which then issues a response indicating the outcome of the bid. The Reo connector

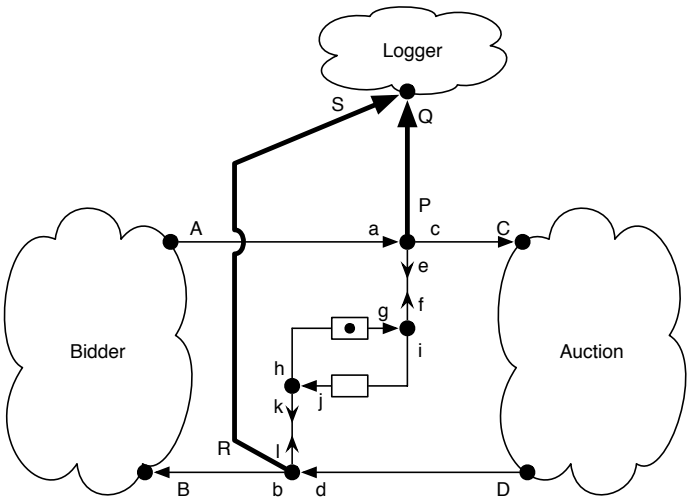


Figure 3. Logging Component added at SQ. All bids and responses are copied to SQ.

guarantees that the simple protocol, alternating bids and responses, is preserved. This scenario has been adapted from the application of Reo to auction protocols [46].

Beyond the initial phases of constructing a connector and connecting the components, a number of reconfiguration scenarios are foreseeable:

- new bidders join an auction and their components are connected;
- bidders leave an auction and their components are disconnected;
- an auction or bidder component is upgraded and replaced;
- the underlying bid-response protocol, enforced by the Reo connector, is modified, for example, to include an authentication phase; or
- a monitoring or logging component is added to the system.

We focus on two particular scenarios: adding logging and changing bidders.

Adding Logging: We want to log all bids and their corresponding responses. This requires the addition of the channels highlighted in reconfigured connector in Figure 3, with the Logging component attached at node SQ. Subsequently, we may remove the logging. We want to reason that: (1) adding logging does not affect the bid-response protocol; (2) all subsequent bids and responses are logged; and (3) removing the logging mechanism produces a connector with the same behaviour as the original.

Changing Bidder: Consider when a bidder (connected to channel ends A and B) leaves an auction and is replaced by another bidder (connected to channel ends A' and B'). With the given bid-response protocol, the following steps are foreseeable: A bid is issued at node A; the reconfiguration occurs to produce the connector in Figure 2(b) with channels A-a and B-b disconnected; and finally, the response corresponding to the original bid is received at node B', which may not be expecting it, instead of at

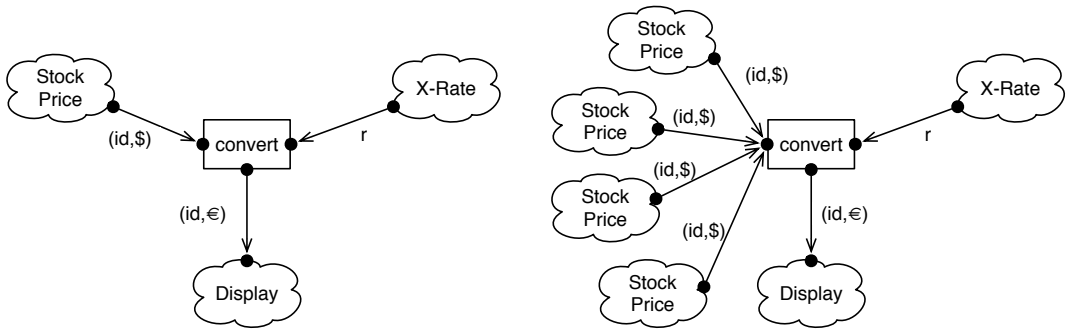


Figure 4. (a) Initial Stock Quote Handler. (b) Reconfigured to handle multiple Stocks.

node B, where it was expected. This scenario could result in incorrect component behaviour including deadlock. In this simple example it is clear that the reconfiguration should only be performed between a response and the subsequent bid. If a party other than the bidder performs the reconfiguration, or if the bidder is not trusted, machinery needs to be added to the connector to avoid a fault in the bid-response protocol. As the bidder may cheat, adding some control may be necessary anyway, but this kind of complication is orthogonal to the issue at hand.

Stock Price Watcher We now consider a slightly more elaborate reconfiguration scenario. Consider the connector in Figure 4(a). This very simple system receives a stock price from a component and an exchange rate from another component, converts the stock price to the desired currency, and passes the resulting value to a display component. As our investment portfolio increases, we will add further stock price components to the system, as in Figure 4(b). Extending the system further in this way will eventually leads to a situation where either the data display shows the items too rapidly to be viewed properly, or the components producing stock prices will need to be delayed, resulting in out-of-date prices being shown. Reconfiguring the system to Figure 5(a), where the lossy syncs channels replace the ordinary synchronous channels, produces a connector that simply discards data values which cannot be displayed in time. Figure 5(b) shows a variant which ensures that stock prices are displayed in a round-robin fashion. The connector uses a *sequencer*, defined by Arbab [2], which alternates data flow at its four nodes. Each reconfiguration step enhances the application and must be performed dynamically, so that no stock price fluctuation is missed. The kind of reasoning one would like to do in this situation is, for example, to ensure that each source of stock prices does actually get to display, that is, that no deadlocks have been introduced through reconfiguration.

It is a simple exercise for the reader to determine the reconfiguration steps required to get from Figure 4(a) to Figure 4(b) to Figure 5(a) to Figure 5(b).

We revisit these scenarios in Section 7.

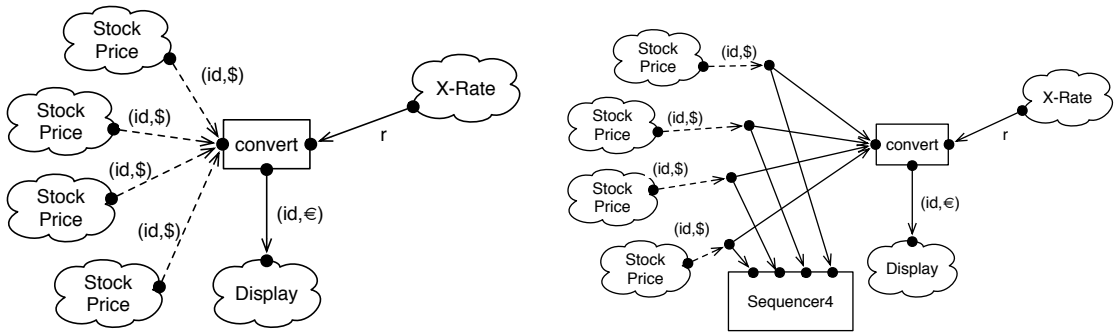


Figure 5. (a) Stock Quote Handler adapted to avoid flooding of data. (b) Adapted further to Schedule Quotes in Round-Robin Fashion.

2. Reo Connectors

This section formalises the "graph" corresponding to a Reo connector in order to precisely describe the structural effect of a reconfiguration operation. A Reo connector is represented as a collection of its constituent channels plus a description of how its ends are grouped to form visible nodes, which can be observed and reconfigured, and hidden and forgotten nodes, which cannot.

Let $\mathcal{E}$ be a denumerable set of channel ends, ranged over by a, b. The function $io : \mathcal{E} \to \{i, o\}$ gives the *direction* of an end: whether it accepts data (*input end*) or produces data (*output end*).[2] A channel is denoted $Ch^{T}_{\mathsf{a},\mathsf{b}}$, where $\mathsf{a}, \mathsf{b} \in \mathcal{E}$ are the ends of the channel, with a and b distinct, and T its type. Each channel type dictates the directionality of each of its ends. For example, synchronous channel $Ch^{Sync}_{\mathsf{f},\mathsf{g}}$ requires that $io(\mathsf{f}) = i$ and $io(\mathsf{g}) = o$.

Connectors are formed by grouping together channel ends into nodes. Thus we represent nodes as sets of channel ends—and thus also consider a single channel end to be a node. Let a, b, c, d, e range over nodes. Let ab denote the joining of nodes a and b, defined as $a \cup b$. *Boundary nodes*, through which components interact with a connector, consist entirely of input ends or entirely of output ends (also called, respectively, *input nodes* and *output nodes*). *Internal* or *mixed* nodes of a connector, indicated by predicate $mixed(a)$, make it possible for data to flow within a connector without any external impetus (see the next section for a description of their behaviour).

The set of nodes in a connector is called its *node set*. The set of visible nodes, those which are not hidden or forgotten, is called its *visible node set*. Let A, B, C range over node sets. H will be reserved for hidden node sets.

Definition 2.1. (Reo Connector)
A Reo connector $\mathcal{C} = (Ch, B, H)$ consists of a set of channels Ch and a set of *visible nodes* B and the *hidden node* set H, where H and B have no channel ends in common. The *node set* $B \cup H$ of the connector satisfies:

1. for all distinct $Ch^{T}_{\mathsf{a},\mathsf{b}}, Ch^{T'}_{\mathsf{c},\mathsf{d}} \in Ch$, a, b, c and d are distinct; and

[2] This terminology differs from Arbab [2] who uses the phrases *source* for *input* and *sink* for *output*. Read *input* as *accepting input* and *output* as *producing output*.

2. $B \cup H$ is a partition of the set of channel ends of channels in Ch.

Denote the collection of all Reo connectors by $\mathcal{R}eo$, ranged over by $\mathcal{C}$ and $\mathcal{D}$.

Example 2.1. The connector in Figure 2(a) is represented by

$$\left(\left\{ \begin{array}{l} Ch_{\mathsf{A,a}}^{Sync}, Ch_{\mathsf{c,C}}^{Sync}, Ch_{\mathsf{e,f}}^{SyncDrain}, Ch_{\mathsf{h,g}}^{FIFO1(\bullet)}, \\ Ch_{\mathsf{i,j}}^{FIFO1}, Ch_{\mathsf{k,l}}^{SyncDrain}, Ch_{\mathsf{b,B}}^{Sync}, Ch_{\mathsf{D,d}}^{Sync} \end{array} \right\}, \{\mathsf{A,B,C,D,ace,gfi,hjk,bdl}\}, \emptyset \right)$$

3. Constructions on Reo Connectors

Reo has a control language for constructing and reconfiguring connectors [2]. It includes operations for creating channels, for joining channel ends to form nodes, for splitting nodes, and for hiding nodes. This control language is, in principle, embedded in a programming language and adopts certain convenient abstractions, such as referring to a node using one if its constituent ends. We present a simplified and clean core of Reo's control language that takes a bird's eye view of reconfiguration. In essence, our language captures the steps performed by (or the trace of) a control language program performing some reconfiguration of a connector. This makes sense, given the context we are working: reconfiguration is performed outside of a connector under the assumption that the behaviour of the connector has no effect on reconfiguration. The traces of the Reo control language are captured in the following language of *constructions*:

$$G \quad ::= \quad \mathtt{id} \mid GG \mid Ch_{\mathsf{a,b}}^{T} \mid \mathtt{join}_{a,b} \mid \mathtt{split}_{a,b} \mid \mathtt{hide}_{a} \mid \mathtt{forget}_{a}$$

Constructions ($\mathcal{C}on$) are (partial) operations taking a Reo connector to another Reo connector (thus their type is $\mathcal{R}eo \to \mathcal{R}eo$). The action of constructions on Reo connectors is given in Definition 3.1. In the prequel [17] to this paper, we explored when constructions are well-formed. The constraints ensured, for example, that the `hide` and `forget` operations were performed on the appropriate kind of node and channel construction avoided creating duplicate names. For this paper, we assume that all constructions satisfy those constraints.

Definition 3.1. (Action of Constructions)
The action of construction $F \in \mathcal{C}on$ on a connector $\mathcal{C} \in \mathcal{R}eo$, denoted $F(\mathcal{C})$, is defined as:

$$\begin{array}{rcll} \mathtt{id}(\mathcal{C}) & = & \mathcal{C} & \\ GF(\mathcal{C}) & = & G(F(\mathcal{C})) & \\ Ch_{\mathsf{a,b}}^{T}(Ch, B, H) & = & (Ch \cup \{Ch_{\mathsf{a,b}}^{T}\}, B \cup \{\mathsf{a}\} \cup \{\mathsf{b}\}, H), & \\ & & \quad \text{where } \{\mathsf{a},\mathsf{b}\} \cap ends(B \cup H) = \emptyset & \\ \mathtt{join}_{a,b}(Ch, B \uplus \{a,b\}, H) & = & (Ch, B \uplus \{ab\}, H) & \\ \mathtt{split}_{a,b}(Ch, B \uplus \{ab\}, H) & = & (Ch, B \uplus \{a,b\}, H) & \\ \mathtt{hide}_{a}(Ch, B \uplus \{a\}, H) & = & (Ch, B, H \uplus \{a\}), & \text{if } mixed(a) \\ \mathtt{forget}_{a}(Ch, B \uplus \{a\}, H) & = & (Ch, B, H \uplus \{a\}), & \text{if } \neg mixed(a) \end{array}$$

where $ends(N)$ is the set of ends underlying a node set N.

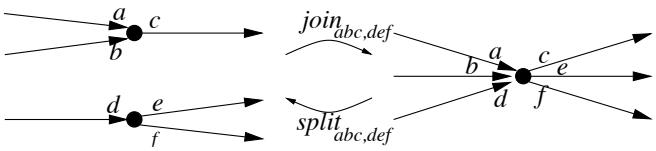


Figure 6. Joining, $\texttt{join}_{abc,def}$, groups nodes abc and def together to form node $abdcef$. Splitting, $\texttt{split}_{abc,def}$, performs the inverse operation. All possible ways of splitting and joining are permitted. Both operations, however, tend to drastically alter data flow.

The identity construction, `id`, does not modify its argument. Sequential composition of F followed by G, is denoted GF, following the mathematical convention. Construction $Ch^T_{\mathsf{a,b}}$ corresponds to creating a new channel of type T with distinct ends $\mathsf{a}, \mathsf{b} \in \mathcal{E}$, and adding the channel ends as unconnected nodes to a connector. Construction $\texttt{join}_{a,b}$ takes two nodes a and b of a connector and joins them together to form a new node ab, and $\texttt{split}_{a,b}$ takes a node ab and splits it into two nodes a and b (see Figure 6).

Construction $\texttt{hide}_a$ makes a mixed node act of its own accord, like a self-contained pumping station performing a pull and push of data whenever it can, independently of behaviour at the boundary, though still observing the constraints imposed by channels and nodes [2]. Hiding is analogous to the standard technique of removing all ϵ-transitions from a non-deterministic finite automaton (or compressing τ-transitions in a process in, say, the π-calculus). The difference between a hidden internal node and a non-hidden internal node is that hidden nodes are required to perform certain transitions before allowing a transition observable to the non-hidden nodes to occur. It must be said, however, that precise understanding of what this entails remains elusive.

Construction $\texttt{forget}_a$ models a boundary node that is no longer in use and thus no longer contributes to the functioning of a connector. No data flows through a forgotten node.

Both $\texttt{hide}_a$ and $\texttt{forget}_a$ have the structural side-effect of preventing a node from being reconfigured, but this is the only behaviour they have in common. We distinguish between hiding and forgetting, because with hiding all possible data flows at the node remain possible, whereas with forgetting all data flow at the node is stopped. From the "real world" perspective, hiding is an operation which is performed to make a connector act as a black box which has no observable internal transitions, whereas forgetting is what happens when a node is simply ignored or dropped from the program and subsequently becomes a candidate for garbage collection.

Example 3.1. The connector in Figure 2(a) is a result of the following construction applied to the empty connector:

$$\begin{aligned}&\texttt{join}_{\mathsf{ac,e}}\texttt{join}_{\mathsf{a,c}}\texttt{join}_{\mathsf{gf,i}}\texttt{join}_{\mathsf{g,f}}\texttt{join}_{\mathsf{hj,k}}\texttt{join}_{\mathsf{h,j}}\texttt{join}_{\mathsf{bd,l}}\texttt{join}_{\mathsf{b,d}}\\ &\quad Ch^{Sync}_{\mathsf{A,a}}\,Ch^{Sync}_{\mathsf{c,C}}\,Ch^{SyncDrain}_{\mathsf{e,f}}\,Ch^{FIFO1(\bullet)}_{\mathsf{h,g}}\,Ch^{FIFO1}_{\mathsf{i,j}}\,Ch^{SyncDrain}_{\mathsf{k,l}}\,Ch^{Sync}_{\mathsf{b,B}}\,Ch^{Sync}_{\mathsf{D,d}}.\end{aligned}$$

The second line creates all the channels; the first line joins ends to form nodes.

Example 3.2. The reconfiguration producing the connector in Figure 3 from the one in Figure 2(a) is:

$$\texttt{join}_{\mathsf{P,ace}}\texttt{join}_{\mathsf{R,bdl}}\texttt{join}_{\mathsf{S,Q}}\,Ch^{Sync}_{\mathsf{R,S}}\,Ch^{Sync}_{\mathsf{P,Q}}.$$

Example 3.3. A construction which takes the connector in Figure 3 and reproduces the connector in Figure 2(a), with some garbage, is:

$$\texttt{forget}_{\texttt{P}}\texttt{forget}_{\texttt{R}}\texttt{forget}_{\texttt{SQ}}\texttt{split}_{\texttt{P,ace}}\texttt{split}_{\texttt{R,bdl}}.$$

4. Constraint Automata: A Semantics for Reo

Constraint automata [7] are an automata-based model where transitions are annotated with (1) a set of names indicating the nodes at which data flows, and (2) a constraint describing the data that flows at the selected nodes. Constraint automata can be used to describe the data flow through nodes and the synchronisation and exclusion constraints on nodes in a Reo connector, where the set of nodes on a transition represents the nodes that are synchronised, and the nodes not present are the excluded nodes. A constraint automaton over visible nodes B has transition labels of the form N, g, where $N \subseteq B$ is the exact, non-empty, set of nodes at which data flows in a step, and g is a data constraint over N describing the data that flows. Data constraints are defined by the following grammar, where $d \in \mathit{Data}$, the data domain:

$$g \quad ::= \quad \mathsf{true} \mid d_a = d \mid d_a = d_b \mid g_1 \wedge g_2 \mid \neg g \mid \exists d_a.g.$$

The data flowing through node a is denoted d_a, thus $d_a = d$ says that the data flowing through node a is d, and $d_a = d_b$ says the data flowing through node a is the same as at node b. The formula $\exists d_a.g$ existentially quantifies over the data flowing at node a in constraint g. Let $DC(B)$ denote the set of all data constraints over visible nodes B, and $DC(N)$ the data constraints over $N \subseteq B$. The precise meaning of data constraints is given by the following definition.

Definition 4.1. (Data Constraint Satisfaction)
Satisfaction of a data constraint g by a data assignment $\delta : \mathit{Node} \rightharpoonup_{\text{fin}} \mathit{Data}$, that is, a finite map from the node names into the data set, is denoted $\delta \models g$ and defined:

$$\begin{array}{llll}
\delta \models \mathsf{true} & \text{always} & \delta \models d_a = d & \iff \delta(a) = d \\
\delta \models d_a = d_b & \iff \delta(a) = \delta(b) & \delta \models g_1 \wedge g_2 & \iff \delta \models g_1 \text{ and } \delta \models g_2 \\
\delta \models \neg g & \iff \delta \not\models g & \delta \models \exists d_a.g & \iff \exists d \in \mathit{Data} \text{ s.t. } \delta[a \mapsto d] \models g.
\end{array}$$

We now define constraint automata [7]:

Definition 4.2. (Constraint Automata)
A *constraint automaton* is a triple $\mathcal{A} = (Q, B, \longrightarrow)$, where Q is a set of states, B is a set of nodes, and $\longrightarrow$ is a subset of $Q \times 2^B \times DC(B) \times Q$, called the transition relation of $\mathcal{A}$. We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. For every *non-trivial* transition, $q \xrightarrow{N,g} p$, we require that (1) $N \neq \emptyset$, and (2) $g \in DC(N)$. In addition, $\longrightarrow$ includes all *trivial* loops $q \xrightarrow{\emptyset,\mathsf{true}} q$ for all $q \in Q$.

Note that a constraint automaton does not give the direction of dataflow, just constraints on the data that flows. The original definition of constraint automata [7] also included the set of initial states. Our presentation has slightly different requirements, so we have removed them and introduced a function

(a) (b)

Figure 7. Example Constraint Automata. (Trivial loops omitted.) We have also abstracted away events on internal nodes of the connector. Whenever data flows through A and C, it flows also through nodes ace and gfi. Similarly, whenever data flows through B and D, it flows also through nodes bdl and hjk.

$\mathit{InitState}(T)$ which gives the initial states of a channel of type T. This gives sufficient information to recover the set of initial states, using the constructions detailed in Section 5. We have also added the notion of *trivial loop* which simplifies the definition of product. Trivial loops play a similar role to ϵ-transitions, also known as stuttering steps, in the definition of synchronous products of labelled transition systems [6].

Arbab *et al* [7] describe how to calculate the constraint automaton for a Reo connector. For connector $\mathcal{C} \in \mathcal{R}eo$, denote the automaton resulting from this construction as $\mathcal{R}[\![\mathcal{C}]\!]$.

Example 4.1. The constraint automaton in Figure 7(a) models the connector in Figure 2(a). It captures the alternating behaviour between synchronous data flow between A and C and between D and B. This matches the expected behaviour of the bid-response protocol. Figure 7(b) is a constraint automaton modelling the behaviour resulting from adding logging at node SQ (cf. Figure 3). The transitions indicate that logging occurs synchronously with both bids and responses, copying the data in both cases.

4.1. Operations on Constraint Automata

Constraint automata are equipped with the operations *product* and *hide* which are used to give the behaviour of connectors in terms of their constituents, and, respectively, of hidden nodes. In addition, we introduce an operation to model the *forget*ting of nodes.

The *product* of two constraint automata with possibly overlapping visible node sets is an automaton which includes the combined behaviour of the constituents such that they agree on the data flowing at the common nodes. Product models, for example, the plugging of an output end in one connector to the input end in another connector.

Definition 4.3. (Product Automata [7])
Given constraint automata $\mathcal{A}_1 = (Q_1, B_1, \longrightarrow_1)$ and $\mathcal{A}_2 = (Q_2, B_2, \longrightarrow_2)$, the product automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$ is $(Q_1 \times Q_2, B_1 \cup B_2, \longrightarrow)$, where $\longrightarrow$ is defined as follows: if $q_1 \xrightarrow{N_1, g_1}_1 p_1$, $q_2 \xrightarrow{N_2, g_2}_2 p_2$, and $N_1 \cap B_2 = N_2 \cap B_1$, then $(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (p_1, p_2)$.

Hiding a node of a constraint automaton produces an automaton in which the behaviour at the node can be performed independently of behaviour at the visible nodes. Let $q \rightsquigarrow_a^* p \in \mathcal{A}$ denote a (possibly

empty) sequence of state transitions of $\mathcal{A}$ starting from state q, ending in state p, involving just node a. That is, $q \rightsquigarrow_a^* p \in \mathcal{A}$ if and only if there exists a finite path in $\mathcal{A}$:

$$q \xrightarrow{\{a\},g_1} q_1 \xrightarrow{\{a\},g_2} q_2 \xrightarrow{\{a\},g_3} \cdots \xrightarrow{\{a\},g_n} p,$$

where each g_i is satisfiable. $q \rightsquigarrow_a^* p$ denotes a sequence of purely internal and non-observable transitions.

Definition 4.4. (Hiding [7])
Given constraint automaton $\mathcal{A} = (Q, B \uplus \{a\}, \longrightarrow)$, the automaton $\texttt{hide}_a(\mathcal{A})$ is $(Q, B, \longrightarrow_a)$, where $\longrightarrow_a$ is defined as follows: if $q \rightsquigarrow_a^* p, p \xrightarrow{N,g} r, N' = N \setminus \{a\} \neq \emptyset$, and $g' = \exists d_a.g$, then $q \xrightarrow{N',g'}_a r$.

The compression of internal data flow that hiding performs (via $\rightsquigarrow_a^*$) does not permit any additional synchronisation beyond what was possible before hiding. For example, data that needs to pass through n-FIFO1 buffers will take 1 step, rather than n. If hiding were to allow this transfer to be instantaneous, then the whole notion of synchrony in Reo would be drastically changed, leading to connectors which would be impossible to realise in practice.

Consult Arbab *et al* [7] for full details of these operations, including examples and correctness proofs.

We now add forgetting to the arsenal of operations on constraint automata. The construction $\texttt{forget}_a$ is applied to boundary nodes at which no further interaction will occur. The forget operation on constraint automata models this by removing all behaviour involving the forgotten node.

Definition 4.5. (Forgetting)
Let $\mathcal{A} = (Q, B \uplus \{a\}, \longrightarrow)$ be a constraint automaton. The constraint automaton $\texttt{forget}_a(\mathcal{A})$ is $(Q, B, \longrightarrow_a)$, where transition relation $\longrightarrow_a$ is given by: if $q \xrightarrow{N,g} p$ and $a \notin N$, then $q \xrightarrow{N,g}_a p$.

Example 4.2. If $\texttt{forget}_A$ is applied to the constraint automaton in Figure 7(a), the result is an automaton with a single non-trivial transition $q \xrightarrow{\{B,D\},d_B=d_D} p$. This captures the fact that no behaviour is possible at node A, and hence also at node C, even though C is still visible.

A note on garbage. In the prequel [17] we determined when certain parts of a circuit corresponded to garbage and we claimed, but did not prove, that removing the garbage caused no problem behaviourally. We state and prove this result here as Theorem 4.1. Consider the graph of a Reo connector. If a connected subgraph of the connector consists entirely of hidden and forgotten nodes, then that subgraph is considered to be garbage, since it can neither produce observable behaviour nor be reconfigured. Let $\mathcal{C} \equiv_{\mathcal{GC}} \mathcal{C}'$ denote that Reo connectors $\mathcal{C}$ and $\mathcal{C}'$ are equivalent modulo garbage.

Theorem 4.1. If $\mathcal{C} \equiv_{\mathcal{GC}} \mathcal{C}'$, then $\mathcal{R}[\![C]\!] \sim \mathcal{R}[\![C']\!]$.

Proof:
A connector $\mathcal{G}$ is garbage if $\mathcal{G} \equiv_{\mathcal{GC}} (\emptyset, \emptyset, \emptyset)$. It is easy to show that $\mathcal{R}[\![\mathcal{G}]\!] \sim \mathbf{0}$, where $\mathbf{0}$ is the constraint automaton with one state and no non-trivial transitions, and $\sim$ is bisimulation as defined by Arbab *et al* [7]. Furthermore, $\mathcal{A} \bowtie \mathbf{0} \sim \mathcal{A}$. Now for two pairs of automata $\mathcal{A}_1 \sim \mathcal{A}_2$ and $\mathcal{B}_1 \sim \mathcal{B}_2$, where the $\mathcal{A}$s are defined over a disjoint node set from the $\mathcal{B}$s, it is easy to show that $\mathcal{A}_1 \bowtie \mathcal{B}_1 \sim \mathcal{A}_2 \bowtie \mathcal{B}_2$. Without loss of generality, assume that $\mathcal{C} = \mathcal{C}' \cup \mathcal{G}$, where $\mathcal{G}$ is garbage. Then $\mathcal{R}[\![\mathcal{C}]\!] = \mathcal{R}[\![\mathcal{C}' \cup \mathcal{G}]\!] \sim \mathcal{R}[\![\mathcal{C}']\!] \bowtie \mathcal{R}[\![\mathcal{G}]\!] \sim \mathcal{R}[\![\mathcal{C}']\!] \times \mathbf{0} \sim \mathcal{R}[\![\mathcal{C}']\!]$. □

This result also enables the reduction of the size of a constraint automaton, which we expect will make model checking (§ 6) more efficient.

5. Reconfiguration Logic — ReCTL*

This section presents the logic ReCTL* for reasoning about reconfiguration. ReCTL* combines the well-known CTL* logic [21] with TSDSL (timed scheduled-data-stream logic) [4][3] for reasoning about Reo connectors (without reconfiguration), and adds a *reconfiguration modality* to express changes in a connector. The time aspect of TSDSL is dropped for simplicity of presentation. Before giving the logic, we introduce the notion of *schedule expression*.

A *schedule expression*, α, is a regular expression of "events":

$$\alpha \quad ::= \quad \langle N, dc\rangle \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1;\alpha_2 \mid \alpha^*$$

Primitive events, $\langle N, dc\rangle$, correspond to data flowing synchronously through the nodes in non-empty set N, where data constraint $dc \in DC(N)$ describes the dataflow. The language of a schedule expression α, denoted $\mathcal{L}(\alpha)$, is defined as [4]:

$$\begin{array}{rclcrcl}
\mathcal{L}(\langle N, dc\rangle) & = & \{\delta \mid \mathrm{dom}(\delta) = N \ \wedge \ \delta \models dc\} & & & & \\
\mathcal{L}(\alpha_1 \vee \alpha_2) & = & \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2) & & \mathcal{L}(\alpha_1 \wedge \alpha_2) & = & \mathcal{L}(\alpha_1) \cap \mathcal{L}(\alpha_2) \\
\mathcal{L}(\alpha_1;\alpha_2) & = & \mathcal{L}(\alpha_1);\mathcal{L}(\alpha_2) & & \mathcal{L}(\alpha^*) & = & \mathcal{L}(\alpha)^*
\end{array}$$

where $L;L' \mathrel{\widehat{=}} \{s.s' \mid s \in L \wedge s' \in L'\}$, $L^0 \mathrel{\widehat{=}} \{\epsilon\}$, $L^{n+1} \mathrel{\widehat{=}} L^n;L$, and $L^* \mathrel{\widehat{=}} \bigcup_{n\geq 0} L^n$.

Example 5.1. The schedule expression $(\langle\{A, C\}, d_A = d_C\rangle;\langle\{B, D\}, d_B = d_D\rangle)^*$ describes that A and C exchange data, then B and D exchange data, zero or more times.

ReCTL* formulæ consist of *state formulæ* ψ and ϕ and *path formulæ*, ρ and ϱ, given by the following grammar, where $a_0 \in \Phi$ are *propositional variables*, and G is any valid construction defined in Section 3:

$$\begin{array}{rcl}
\psi, \phi & ::= & \mathsf{true} \mid a_0 \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \mathsf{E}\rho \mid \langle G\rangle\psi \\
\rho, \varrho & ::= & \psi \mid \rho_1 \wedge \rho_2 \mid \neg\rho \mid \langle\!\langle\alpha\rangle\!\rangle\rho \mid \rho_1 \,\mathsf{U}\, \rho_2
\end{array}$$

Modalities $\mathsf{E}\rho$ and $\rho_1 \,\mathsf{U}\, \rho_2$ are standard from CTL* [21], stating, respectively, that there is some sequence of transitions (a run) in the automaton which will result in a state satisfying ρ, and, for a given run of the automaton, ρ_1 holds up to some point, after which ρ_2 holds. The modality $\langle\!\langle\alpha\rangle\!\rangle\rho$ states that a path has a prefix contained in $\mathcal{L}(\alpha)$ whose subsequent behaviour satisfies ρ. This modality has been adapted from TSDSL [4]. Its dual, $[\![\alpha]\!]\rho \mathrel{\widehat{=}} \neg\langle\!\langle\alpha\rangle\!\rangle\neg\rho$, states that the subsequent behaviour for all prefixes of the path matching α satisfies ρ. $\langle\!\langle\alpha\rangle\!\rangle-$ obviates the need for CTL*'s $\mathsf{X}-$ modality, as $\langle\!\langle \bigvee_{N_0 \subseteq N, N_0 \neq \emptyset} \langle N_0, \mathsf{true}\rangle\rangle\!\rangle-$ does the trick, where N is the visible node set of the connector in question. For each construction G, the modality $\langle G\rangle\psi$ states that ψ holds in some state of the connector resulting from the reconfiguration G. The dual $[\![G]\!]\psi \mathrel{\widehat{=}} \neg\langle G\rangle\neg\psi$ states that ψ holds in all such states.

[3] As pointed out by an anonymous referee, RCTL [9] may have been a better starting point for our logic, as it resembles the time-free fragment of TSDSL in a CTL setting rather than an LTL setting.

The logic (excluding the reconfiguration modality) can be seen as extending CTL*, as TSDSL extended LTL, with $\langle\!\langle\alpha\rangle\!\rangle-$ replacing $\mathsf{X}-$ to reason about transition labels. The above collection of operators was selected for a number of reasons. Firstly, the CTL* operators are well-known for reasoning about branching behaviour, and their inclusion should not be contentious. In addition, CTL*'s model proved easier to add the reconfiguration modality to than an LTL-like model. In fact, CTL* proved to be more suitable as a base than TSDSL to add the reconfiguration modality to, and the semantics and model checking of the resulting logic are quite natural. Secondly, TSDSL's modality enables reasoning about details of the data flow, compared to $\mathsf{X}-$ of CTL*, which permits reasoning only about the fact that a step has occurred, without expressing anything about the details. This change makes the logic similar to ACTL [34], except that $\langle\!\langle\alpha\rangle\!\rangle-$ permits reasoning about whole languages of actions, rather than just a single action. Lastly, the reconfiguration modality naturally models reconfigurations which are initiated from outside the connector being reconfigured. It directly captures the change in connector, and remains non-deterministic with regard to the exact state the new connector is in.

Example 5.2. Here are some formulæ and their interpretations:

1. The sequence expression $(\bigvee_{N'\subseteq N, N'\neq\emptyset}\langle N', \mathsf{true}\rangle)^*$, denoted by $\star$, where N is the nodes of the automaton in question, denotes an arbitrary (finite) sequence of events. We will use this often in formulæ.
2. $\mathsf{A}[[\star]]\psi$ states that ψ is true in all states of a connector/constraint automaton.
3. $\psi \to [\![G]\!]\psi$ states that reconfiguration G preserves the property ψ.
4. $\psi \to \langle\!\langle G\rangle\!\rangle\psi$ states that reconfiguration G may preserve the property ψ.
5. $\psi \to [[\star]][\![G]\!]\psi$ states that after any number of steps of the initial connector, reconfiguration G still preserves ψ.
6. $\mathsf{A}[[\star]][\![G]\!][[\star]]\mathsf{X}\mathsf{true}$ states that reconfiguration G never leads to deadlock.
7. $\mathsf{A}[[(X;Y)^*]]\langle\!\langle X\rangle\!\rangle\mathsf{true}\wedge\mathsf{A}[[(X;Y)^*;X]]\langle\!\langle Y\rangle\!\rangle\mathsf{true}$ states that observations X and Y alternate, starting with X.
8. $\mathsf{A}[[\star]][\![G]\!](\mathsf{A}[[(X;Y)^*]]\langle\!\langle X\rangle\!\rangle\mathsf{true} \wedge \mathsf{A}[[(X;Y)^*;X]]\langle\!\langle Y\rangle\!\rangle\mathsf{true})$ states that it is always the case that, after any number of steps, reconfiguration G will result in a connector in which X and Y alternate, starting with X.

More examples, involving time but not reconfiguration, can be found in Arbab *et al*. [4].

We expect that the logic will be useful for at least the following applications: specifying that data flows observing some protocol, and then continues to follow the same protocol, perhaps in a larger connector; specifying that data flows observing some protocol, and then follows some alternative protocol after reconfiguration; specifying that reconfiguration does not lead to deadlock; specifying that some states are safe to perform reconfiguration in, whereas others are not; specifying the changed routing of data due to reconfiguration; and so forth.

On the other hand, the logic is not expressive enough to deal with situations that require quantification over a collection of connectors, such as asking questions like "for any connector satisfying ψ, it is

the case that ϕ holds" and "if a connector enforces ψ, then the reconfigured connector enforces ϕ", or for quantifying over the reconfiguration operation. The main difficulty here is that reconfiguration can typically distinguish different connectors that have identical behaviour. In any case, such questions are beyond the scope of our paper, as our primary aim is to develop a logic for reasoning about a particular connector under specific reconfiguration scenario(s).

Given that connectors can become arbitrarily complex, certainly more so than our simple examples, model checking is the only feasible approach to determining the correctness of formulæ describing the expected behaviour of a given connector in a given reconfiguration scenario.

5.1. Semantics of ReCTL*

We now present the semantics of ReCTL* formulæ in terms of runs of a constraint automaton and transitions between automata. This can also be seen as the semantics of Reo in the presence of reconfiguration. Two kinds of behaviour are possible: firstly, input and output can be performed within a given connector — this is modelled as a state transition *within* the appropriate constraint automaton (Definition 5.1); and secondly, a reconfiguration step can be performed — this is modelled as a reconfiguration transition *between* automata (Definition 5.4). The semantics can thus be seen as a graph of constraint automata, with constraint automata at the vertices and reconfiguration operations labelling the edges.

The semantics are based on the notion of a run, which is a sequence of state transitions.

Definition 5.1. (State Transition)
A *state transition* for a constraint automaton $\mathcal{A}$ is given by $q \xrightarrow{\delta} q'$, where δ is a data assignment from some non-empty set N to $Data$ for which there is a transition $q \xrightarrow{N,g} q' \in \mathcal{A}$ satisfying $\delta \models g$.
Let $ST(\mathcal{A})$ denote the set of state transitions for constraint automaton $\mathcal{A}$.

Observe that a state transition is labelled with a solution to the constraints of some transition in the constraint automaton, thus the two kinds of transition are different notions.

Definition 5.2. (Run)
A *q-run* of a constraint automaton $\mathcal{A}$ is a finite or infinite sequence: $\pi = q_0 \xrightarrow{\delta_0} q_1 \xrightarrow{\delta_1} \cdots$, where $q_0 = q$ and each $q_i \xrightarrow{\delta_i} q_{i+1} \in ST(\mathcal{A})$ is a state transition.

The *first state of a q-run* is, by definition, q. Let π^i denote the suffix of π starting at i:

$$\pi^i \mathrel{\widehat{=}} q_i \xrightarrow{\delta_i} q_{i+1} \xrightarrow{\delta_{i+1}} \cdots .$$

Let $\pi{\downarrow_j}$ be the sequence of labels of the prefix of π preceding the jth element, defined as:

$$\pi{\downarrow_0} \mathrel{\widehat{=}} \epsilon \qquad\qquad (q \xrightarrow{\delta} \pi){\downarrow_{j+1}} \mathrel{\widehat{=}} \delta.\pi{\downarrow_j} .$$

Whenever reconfiguration occurs, a new connector results. The state of the original connector, such as the contents of FIFO buffers, is preserved by the reconfiguration, as reconfiguration affects only the connections between channels, not the channels themselves. To capture this formally, we define the function $\mathcal{S}_{G,\mathcal{C}}(-)$ which maps each state of the automaton for connector $\mathcal{C}$ to the set of states to which it corresponds after performing reconfiguration step G. There is one caveat to this description: when a node is hidden, it may initiate behaviour of its own accord. The definition of state transfer for $\texttt{hide}_a$ takes this behaviour into account.

Definition 5.3. (State Transfer)
The state transfer function for applying construction G to connector $\mathcal{C}$, $\mathcal{S}_{G,\mathcal{C}}(-) : \mathit{State}_{\mathcal{C}} \to \mathcal{P}(\mathit{State}_{G(\mathcal{C})})$, is defined:

$$\begin{array}{lcllcllcl} \mathcal{S}_{Ch^{T}_{\mathtt{a},\mathtt{b}},\mathcal{C}}(q) & = & \mathit{InitStates}(T) \times \{q\} & \mathcal{S}_{\mathtt{id},\mathcal{C}}(q) & = & \{q\} & \mathcal{S}_{\mathtt{join}_{a,b},\mathcal{C}}(q) & = & \{q\} \\ \mathcal{S}_{FG,\mathcal{C}}(q) & = & \mathcal{S}_{F,G(\mathcal{C})}(\mathcal{S}_{G,\mathcal{C}}(q)) & \mathcal{S}_{\mathtt{split}_{a,b},\mathcal{C}}(q) & = & \{q\} & \mathcal{S}_{\mathtt{forget}_{a},\mathcal{C}}(q) & = & \{q\} \\ & & & \mathcal{S}_{\mathtt{hide}_{a},\mathcal{C}}(q) & = & \{p \mid q \rightsquigarrow_{a}^{*} p \in \mathcal{R}[\![\mathcal{C}]\!]\}. & & & \end{array}$$

where $\mathit{State}_{\mathcal{C}}$ denotes the states of the constraint automaton underlying connector $\mathcal{C}$ and $\mathit{InitStates}(T)$ contains the initial states of a channel of type T.

Note that the first clause of this definition produces ordered pairs of states, which is in line with the definition of product on constraint automata.

This definition can be lifted to sets of states to obtain:

$$\mathcal{S}_{G,\mathcal{C}}(-) : \mathcal{P}(\mathit{State}_{\mathcal{C}}) \to \mathcal{P}(\mathit{State}_{G(\mathcal{C})}) = \{p \in \mathcal{S}_{G,\mathcal{C}}(q) \mid q \in Q\}.$$

State transfer functions are sensible in at least the following sense:

Lemma 5.1. $\mathcal{S}_{F,\mathcal{C}}(\mathit{State}_{\mathcal{C}}) \subseteq \mathit{State}_{F(\mathcal{C})}$.

Proof:
Apart from product, each operation on constraint automata preserves the state set of an automaton, so the corresponding clause of $\mathcal{S}_{F,\mathcal{C}}(-)$ does too. The product operation on constraint automata is used to model the addition of a new channel to a connector. The state set of a product automaton is the cartesian product of the state set of its two argument automata. So, given that $\mathit{InitStates}(T)$ is a subset of the first argument automaton's state set and that $\{q\}$ is a subset of the second argument automaton's states, it is clear that $\mathcal{S}_{Ch^{T}_{\mathtt{a},\mathtt{b}},\mathcal{C}}(q) = \mathit{InitStates}(T) \times \{q\}$ is a subset of the product automaton's state set. □

The model underlying the semantics of ReCTL* has two forms (one for each kind of formula). The first form is $\langle \mathcal{C}, \mathcal{A}, V, q \rangle$, where $\mathcal{C}$ is a Reo connector, $\mathcal{A} = \mathcal{R}[\![\mathcal{C}]\!] = \langle Q, B, \rightarrow \rangle$ is the constraint automaton of the connector $\mathcal{C}$, $V : \Phi \to \mathcal{P}(Q)$ is a valuation mapping propositional variables into subsets of the state set Q, and $q \in Q$ is a state of the constraint automaton. The second form is $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle$, where π is a run of the constraint automaton. Note that the Reo connector $\mathcal{C}$ is required in the model, as reconfiguration drastically changes the behaviour of connectors in a manner which cannot be computed compositionally using just constraint automaton — the automaton needs to be recomputed.

We can now introduce a transition relation between models which describes the changes resulting from reconfiguration. We reiterate: given that a constraint automaton captures the semantics of a Reo connector, the reconfiguration transition provides the semantics of reconfiguration.

Definition 5.4. (Reconfiguration Transition)
A reconfiguration transition between two models for reconfiguration operation G, denoted

$$\langle \mathcal{C}, \mathcal{A}, V, q \rangle \xrightarrow{G} \langle \mathcal{C}', \mathcal{A}', V', q' \rangle,$$

is defined iff (i) $\mathcal{C}' = G(\mathcal{C})$ is defined, (ii) $\mathcal{A}' = \mathcal{R}[\![G(\mathcal{C})]\!] = (Q', B', \rightarrow')$, (iii) $q' \in \mathcal{S}_{G,\mathcal{C}}(q)$, and (iv) $V' = \mathcal{S}_{G,\mathcal{C}}(V)$, where $\mathcal{S}_{G,\mathcal{C}}(-)$ is extended to transfer valuations across models as follows $\mathcal{S}_{G,\mathcal{C}}(V)(a_0) = \mathcal{S}_{G,\mathcal{C}}(V(a_0))$.

$$
\begin{array}{lcl}
\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \mathsf{true} & & \text{always} \\
\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models a_0 & \iff & q \in V(a_0) \\
\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \psi_1 \wedge \psi_2 & \iff & \langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \psi_1 \text{ and } \langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \psi_2 \\
\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \neg\psi & \iff & \langle \mathcal{C}, \mathcal{A}, V, q\rangle \not\models \psi \\
\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \langle\!\langle G\rangle\!\rangle\psi & \iff & \exists \mathcal{C}', \mathcal{A}', V', q' \text{ s.t. } \langle \mathcal{C}, \mathcal{A}, V, q\rangle \xrightarrow{G} \langle \mathcal{C}', \mathcal{A}', V', q'\rangle \\
 & & \quad \text{and } \langle \mathcal{C}', \mathcal{A}', V', q'\rangle \models \psi \\
\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \mathsf{E}\rho & \iff & \text{there is a } q\text{-run } \pi \text{ in } \mathcal{A} \text{ s.t. } \langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho \\
\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \psi & \iff & p \text{ is the first state of } \pi \text{ and } \langle \mathcal{C}, \mathcal{A}, V, p\rangle \models \psi \\
\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho_1 \wedge \rho_2 & \iff & \langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho_1 \text{ and } \langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho_2 \\
\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \neg\rho & \iff & \langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \not\models \rho \\
\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \langle\!\langle \alpha\rangle\!\rangle\rho & \iff & \text{there exists } i \geq 0 \text{ s.t. } \pi{\downarrow_i} \in \mathcal{L}(\alpha) \text{ and } \langle \mathcal{C}, \mathcal{A}, V, \pi^i\rangle \models \rho \\
\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho_1 \,\mathsf{U}\, \rho_2 & \iff & \text{there exists } k \geq 0 \text{ s.t. } \langle \mathcal{C}, \mathcal{A}, V, \pi^k\rangle \models \rho_2 \text{ and} \\
 & & \quad \langle \mathcal{C}, \mathcal{A}, V, \pi^j\rangle \models \rho_1 \text{ for all } 0 \leq j < k
\end{array}
$$

Figure 8. Semantics of ReCTL*.

Reconfiguration G results in (i) a new connector, (ii) a recomputed constraint automaton, and (iii) one of the possible states in which this automaton could be. Finally, (iv) maps the valuation into the states of the new automaton, required for the semantics of the logic.

Finally, we give the semantics of ReCTL* in Figure 8. The semantics is based on two relations $\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \psi$, where ψ is a state formula, and $\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho$, where ρ is a path formula.

6. Model Checking

ReCTL* is used to describe desired properties of connectors in the presence of zero or more reconfiguration operations. The goal of model checking, therefore, is to check whether the property holds for a given connector (in a given state). The model checking algorithm computes the following: Given a *connector* $\mathcal{C}$, which states of its underlying automaton satisfy state formula ψ? Namely,

$$\mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi) \quad = \quad \{q \in \mathit{States}(\mathcal{A}) \mid \langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \psi\}$$

where $\mathit{State}(\mathcal{A})$ is the set of states of the constraint automaton $\mathcal{A}$. Thus the model checking algorithm gives all states in which the property holds. Typically, we want to ensure that the initial states of the connector belong to this set.

The model checking algorithm for ReCTL* derives from those of CTL* [21] and TSDSL [4]. The major novelty is the checking of $\langle\!\langle G\rangle\!\rangle\psi$. First we assume that the data domain, *Data*, is finite and, hence, that all quantifiers in data constraints are replaced by finite disjunctions or conjunctions over *Data*.

6.1. Model Checking time-free TSDSL

The model checking algorithm for ReCTL* relies on a model checking algorithm for the following fragment of ReCTL*, just as model checking CTL* can depend on model checking LTL [21]. This fragment is the $\mathsf{E}-$ and $\langle G \rangle -$ free fragment of ReCTL*, which corresponds to the time-free fragment of TSDSL [4]:

$$\rho^\dagger, \varrho^\dagger \quad ::= \quad \mathsf{true} \mid a_0 \mid \rho_1^\dagger \wedge \rho_2^\dagger \mid \neg\rho^\dagger \mid \langle\!\langle \alpha \rangle\!\rangle \rho^\dagger \mid \rho_1^\dagger \,\mathsf{U}\, \rho_2^\dagger.$$

The model checking algorithm for TSDSL [4] can be adapted to compute the following:

$$\mathcal{MC}_{TSDSL}(\mathcal{C}, \mathcal{A}, V, \mathsf{A}\rho^\dagger) \quad = \quad \{q \in States(\mathcal{A}) \mid \forall \pi, \text{a } q\text{-run of } \mathcal{A}, \langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho^\dagger\}.$$

This algorithm clearly satisfies the following property:

Lemma 6.1. ([4])
$\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \mathsf{A}\rho^\dagger$ if and only if $q \in \mathcal{MC}_{TSDSL}(\mathcal{C}, \mathcal{A}, V, \mathsf{A}\rho^\dagger)$.

6.2. Model Checking ReCTL*

Figure 9 presents an algorithm for computing $\mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi)$. It recurses the structure of ψ in a straightforward manner. This approach enables a simpler proof of correctness than for more imperative algorithms, but, as is, it does not serve as a good basis for an implementation. Note that $\mathcal{MC}_{ReCTL^*}$ appears not to use $\mathcal{A}$; it is, however, probed in the $\mathcal{MC}_{TSDSL}(-)$ subroutine described in the previous subsection.

Checking $\langle G \rangle \psi$ proceeds as follows. Firstly, a jump is made into the state space of the new connector using the state transition function defined above (Definition 5.3). The connector, constraint automaton and valuation are updated accordingly. The change to the valuation reflects which states of the new automaton correspond to true states for each atomic proposition a_0 in the initial valuation. Next, ψ is model checked to determine the states of the reconfigured connector in which it holds. The function

$$\mathcal{S}^{\cap}_{G,\mathcal{C}}(Q) \;:\; \mathcal{P}(State_{G(\mathcal{C})}) \to \mathcal{P}(State_{\mathcal{C}}) = \{q \in State_{\mathcal{C}} \mid \mathcal{S}_{G,\mathcal{C}}(q) \cap Q \neq \emptyset\}$$

is then applied to the resulting state set to map it back into the state set of the original connector. This function is an inverse of the state transfer function appropriate for checking a "possibility" style modality. The intuition is as follows: if Q corresponds to the states at which ψ holds in the reconfigured model, then $\mathcal{S}^{\cap}_{G,\mathcal{C}}(Q)$ is the set of states of the original model which possibly map into Q via the action of the reconfiguration G. (This function gives a quite natural treatment of the possibility modality. It corresponds to rough sets' *upper representation* operation when applied between two separate models, rather than a single model [42]. Work by Yao and Lin [44] connects the upper representation with possibility modalities in the context of a single model. So our function is the natural generalisation to separate models.)

The standard technique for checking formulæ of the form $\mathsf{E}\rho$ is used [21]. The idea is to reduce ρ to a TSDSL formula by replacing each $\mathsf{E}\varrho$ and, additionally each $\langle G \rangle \psi$, by a fresh atom, and extending the valuation to map this atom to the states in which the replaced formula ($\mathsf{E}\varrho$ or $\langle G \rangle \psi$) holds. The function $\mathcal{E}lim(-)$ performs the desired operation, mapping an ReCTL* formula to a TSDSL formula and valuation, which are then fed into $\mathcal{MC}_{TSDSL}(-)$.

$$
\begin{array}{rcl}
\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\mathsf{true}) & = & \mathit{States}(\mathcal{A}) \\
\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,a_0) & = & V(a_0) \\
\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\psi_1 \wedge \psi_2) & = & \mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\psi_1) \cap \mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\psi_2) \\
\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\neg\psi) & = & \overline{\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\psi)} \\
\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\langle G\rangle\psi) & = & \mathcal{S}^{\cap}_{G,\mathcal{C}}(\mathcal{MC}_{ReCTL^*}(G(\mathcal{C}),\mathcal{R}[\![G(\mathcal{C})]\!],\mathcal{S}_{G,\mathcal{C}}(V),\psi)) \\
\mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\mathsf{E}\rho) & = & \overline{\mathcal{MC}_{TSDSL}(\mathcal{C},\mathcal{A},V\cup V',\mathsf{A}\neg\rho')}, \\
 & & \quad \mathit{where}\ (\rho',V') = \mathcal{E}\mathit{lim}(\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\text{where}\quad \mathcal{E}\mathit{lim}(\mathsf{true}) & = & (\mathsf{true},\emptyset) \\
\mathcal{E}\mathit{lim}(a_0) & = & (a_0,\emptyset) \\
\mathcal{E}\mathit{lim}(\psi_1\wedge\psi_2) & = & (\psi_1'\wedge\psi_2', V_1\cup V_2), \\
 & & \quad \mathit{where}\ (\psi_1',V_1) = \mathcal{E}\mathit{lim}(\psi_1)\ \mathit{and}\ (\psi_2',V_2) = \mathcal{E}\mathit{lim}(\psi_2) \\
\mathcal{E}\mathit{lim}(\neg\psi) & = & (\neg\psi', V'),\ \mathit{where}\ (\psi',V') = \mathcal{E}\mathit{lim}(\psi) \\
\mathcal{E}\mathit{lim}(\langle G\rangle\psi) & = & (a_0,\{a_0\mapsto \mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\langle G\rangle\psi)\}),\ \mathit{where}\ a_0\ \mathit{is\ fresh} \\
\mathcal{E}\mathit{lim}(\mathsf{E}\rho) & = & (a_0,\{a_0\mapsto \mathcal{MC}_{ReCTL^*}(\mathcal{C},\mathcal{A},V,\mathsf{E}\rho)\}),\ \mathit{where}\ a_0\ \mathit{is\ fresh} \\
\mathcal{E}\mathit{lim}(\langle\!\langle\alpha\rangle\!\rangle\rho) & = & (\langle\!\langle\alpha\rangle\!\rangle\rho', V'),\ \mathit{where}\ (\rho',V') = \mathcal{E}\mathit{lim}(\rho) \\
\mathcal{E}\mathit{lim}(\rho_1\,\mathsf{U}\,\rho_2) & = & (\rho_1'\,\mathsf{U}\,\rho_2', V_1\cup V_2), \\
 & & \quad \mathit{where}\ (\rho_1',V_1) = \mathcal{E}\mathit{lim}(\rho_1)\ \mathit{and}\ (\rho_2',V_2) = \mathcal{E}\mathit{lim}(\rho_2)
\end{array}
$$

Figure 9. Model Checking ReCTL*. $\overline{S} = \mathit{States}(\mathcal{A}) \setminus S$ is the complement of state set S with respect to automaton $\mathcal{A}$, determined from context.

6.3. Properties

The graph of constraint automata that forms the basis of our model is infinite, as reconfiguration operations can be applied to construct any possible connector. This is not problematic for model checking, because only a finite number of reconfiguration steps can appear in a formula, and thus only a finite number of constraint automata needs to be explored. On-the-fly model checking deals with infinite state models in a similar manner as we do [32]. Model checking within a given constraint automaton is bounded (though Arbab *et al* [4] present no complexity results we can draw from), and thus our model checking algorithm is also bounded. We estimate that the complexity is roughly *the number of reconfiguration operations* $\times$ *the cost of constructing a constraint automata from a Reo connector* $\times$ *the cost of CTL* model checking*. The cost of constructing a constraint automaton is bounded by the product of the number of transitions in the constraint automata for its constituent channels [28], which is at worst exponential in the size of a connector.

In any case, we have argued that the the following property holds:

Lemma 6.2. The model checking problem for ReCTL* is decidable.

The model checking algorithm satisfies the following properties, the second of which is correctness.

Lemma 6.3. If $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho$ and $a_0 \notin \mathrm{dom}(V)$, then $\langle \mathcal{C}, \mathcal{A}, V \cup \{a_0 \mapsto Q\}, \pi \rangle \models \rho$, for all $Q \subseteq \mathit{States}(\mathcal{A})$.

Lemma 6.4. 1. $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \psi$ if and only if $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi)$.

2. $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho$ if and only if $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi \rangle \models \rho'$, where $(\rho', V') = \mathcal{E}\mathit{lim}(\rho)$.

Proof:
Proof is by mutual induction. Part 1 proceeds by case analysis on ψ.

1. Cases true and a_0: immediate.

2. Case $\psi_1 \wedge \psi_2$: $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \psi_1 \wedge \psi_2$ if and only if $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \psi_1$ and $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \psi_2$. By induction hypothesis, this is equivalent to $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi_1)$ and $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi_2)$, and hence to $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi_1) \cap \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi_2)$, and finally to $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi_1 \wedge \psi_2)$.

3. Case $\neg\psi$: $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \neg\psi$ if and only if $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \not\models \psi$. By induction hypothesis, this is equivalent to $q \notin \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \psi)$, which is equivalent to $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \neg\psi)$.

4. Case $\mathsf{E}\rho$: $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \mathsf{E}\rho$ if and only if there is a q-run π such that $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho$. This holds by part 2 of this lemma, if and only if $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi \rangle \models \rho'$, where $\mathcal{E}\mathit{lim}(\rho) = (V', \rho')$. Thus the initial premise is equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V', q \rangle \models \mathsf{E}\rho'$. Observing that ρ' is both $\mathsf{E}-$ and $\langle G \rangle -$ free, this is, by Lemma 6.1, equivalent to $q \in \mathcal{MC}_{\mathit{TSDSL}}(\mathcal{C}, \mathcal{A}, V \cup V', \mathsf{E}\rho')$, which is equivalent to $q \notin \mathcal{MC}_{\mathit{TSDSL}}(\mathcal{C}, \mathcal{A}, V \cup V', \mathsf{A}\neg\rho')$, which is in turn equivalent to $q \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}, \mathcal{A}, V, \mathsf{E}\rho)$.

5. Case $\langle G \rangle \psi$: $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \langle G \rangle \psi$ if and only if, by Definition 5.4, there is a $q' \in \mathcal{S}_{G,\mathcal{C}}(q)$ such that $\langle \mathcal{C}', \mathcal{A}', V', q' \rangle \models \psi$, where $\mathcal{C}' = G(\mathcal{C})$, $\mathcal{A}' = \mathcal{R}[\![G(\mathcal{C})]\!]$ and $V' = \mathcal{S}_{G,\mathcal{C}}(V)$. By induction hypothesis, this is equivalent to $q' \in \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}', \mathcal{A}', V', \psi)$. Setting $Q = \mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}', \mathcal{A}', V', \psi)$, we have shown that $q' \in \mathcal{S}_{G,\mathcal{C}}(q)$ and $q' \in Q$, which is equivalent to $q' \in \{q \in \mathit{State}_{\mathcal{C}} \mid \mathcal{S}_{G,\mathcal{C}}(q) \cap Q \neq \emptyset\}$. By definition, we obtain that this is equivalent to $q' \in \mathcal{S}^{\cap}_{G,\mathcal{C}}(\mathcal{MC}_{\mathit{ReCTL}^*}(\mathcal{C}', \mathcal{A}', V', \psi))$, and hence that $q' \in \mathcal{MC}_{\mathit{ReCTL}^*}(G(\mathcal{C}), \mathcal{C}, \mathcal{A}, V, \langle G \rangle \psi)$.

Proof of part 2, by case analysis on ρ. Note that a state formula can also be a path formula, so induction is also over ψ:

1. Cases true and a_0: immediate.

2. Case $\psi_1 \wedge \psi_2, \rho_1 \wedge \rho_2$: $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho_1 \wedge \rho_2$ if and only if $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho_1$ and $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho_2$. By induction hypothesis, this is equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V_1, \pi \rangle \models \rho_1'$ and $\langle \mathcal{C}, \mathcal{A}, V \cup V_2, \pi \rangle \models \rho_2'$, where $(\rho_1', V_1) = \mathcal{E}\mathit{lim}(\rho_1)$ and $(\rho_2', V_2) = \mathcal{E}\mathit{lim}(\rho_2)$. By two applications of Lemma 6.3, this is equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V_1 \cup V_2, \pi \rangle \models \rho_1'$ and $\langle \mathcal{C}, \mathcal{A}, V \cup V_1 \cup V_2, \pi \rangle \models \rho_2'$, as the domain of V_1 and V_2 are disjoint due to the freshness assumptions. This is now equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V_1 \cup V_2, \pi \rangle \models \rho_1' \wedge \rho_2'$, where $(\rho_1' \wedge \rho_2', V_1 \cup V_2) = \mathcal{E}\mathit{lim}(\rho_1 \wedge \rho_2)$.

3. Case $\neg\psi, \neg\rho$: straightforward application of induction hypothesis.

4. Case $\mathsf{E}\rho$: $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \mathsf{E}\rho$ if and only if $\langle \mathcal{C}, \mathcal{A}, V, q \rangle \models \mathsf{E}\rho$, where q is the first state of run π. By part 1 of this lemma, this is equivalent to $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \mathsf{E}\rho)$.[4] Now given $V' = \{a_0 \mapsto \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \mathsf{E}\rho)\}$, where a_0 is fresh, $\langle \mathcal{C}, \mathcal{A}, V \cup V', q \rangle \models a_0$ if and only if $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi \rangle \models a_0$, as π is a q-run. Recalling that $\mathcal{E}lim(\mathsf{E}\rho) = (a_0, V')$, we are done.

5. Case $\langle\!\langle G \rangle\!\rangle \psi$: essentially the same argument as for $\mathsf{E}\rho$.

6. Case $\langle\!\langle \alpha \rangle\!\rangle \rho$: $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \langle\!\langle \alpha \rangle\!\rangle \rho$ if and only if there exists $i \geq 0$ such that $\pi{\downarrow_i} \in \mathcal{L}(\alpha)$ and $\langle \mathcal{C}, \mathcal{A}, V, \pi^i \rangle \models \rho$. By the induction hypothesis, this is equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi^i \rangle \models \rho'$, where $(\rho', V') = \mathcal{E}lim(\rho)$. Hence the original supposition is equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi \rangle \models \langle\!\langle \alpha \rangle\!\rangle \rho'$.

7. Case $\rho_1 \mathsf{U} \rho_2$: $\langle \mathcal{C}, \mathcal{A}, V, \pi \rangle \models \rho_1 \mathsf{U} \rho_2$ if and only if there exists $k \geq 0$ such that $\langle \mathcal{C}, \mathcal{A}, V, \pi^k \rangle \models \rho_2$ and $\langle \mathcal{C}, \mathcal{A}, V, \pi^j \rangle \models \rho_1$ for all $0 \leq j < k$. By the induction hypothesis and Lemma 6.3, as used in the case for $\rho_1 \wedge \rho_2$, this is equivalent to $\langle \mathcal{C}, \mathcal{A}, V \cup V_1 \cup V_2, \pi^k \rangle \models \rho_2'$ and $\langle \mathcal{C}, \mathcal{A}, V \cup V_1 \cup V_2, \pi^j \rangle \models \rho_1'$ for all $0 \leq j < k$, where $(\rho_1', V_1) = \mathcal{E}lim(\rho_1)$ and $(\rho_2', V_2) = \mathcal{E}lim(\rho_2)$. Finally, this is equivalent to the desired result $\langle \mathcal{C}, \mathcal{A}, V \cup V_1 \cup V_2, \pi \rangle \models \rho_1' \mathsf{U} \rho_2'$, where $(\rho_1' \mathsf{U} \rho_2', V_1 \cup V_2) = \mathcal{E}lim(\rho_1 \mathsf{U} \rho_2)$.

□

Note on Counterexample Generation For model checking to be more useful, rather than simply giving the states of the connector which satisfy the formula, the algorithm ought to also return some kind of counterexample for the states that do not satisfy the formula. It is beyond the scope of this paper to give a detailed account of counterexample generation. Standard approaches for CTL and variants can be adapted to give counterexample for the cases without reconfiguration [20]. Such a counterexample will begin with a particular state of the connector resulting from the reconfiguration. Using the function $\mathcal{S}^{\cap}_{G,\mathcal{C}}(q)$ on such a state would enable the construction of the counterexample to continue in the context of the initial connector.

7. Reconfiguration Scenarios, Revisited

Auction ReCTL* can be used to describe the behaviour of a fixed connector as in Arbab *et al* [4]. For Figure 2(a), let $\mathbf{A} = \langle \{A, C\}, d_A = d_C \rangle$ denote the event data flowing synchronously from A to C, and $\mathbf{B} = \langle \{B, D\}, d_B = d_D \rangle$ denote the synchronous flow from D to B. Similarly, for Figure 2(b) (changing bidders), let $\mathbf{A'}$ and $\mathbf{B'}$ denote analogous events in the reconfigured connector. For Figure 3 (adding logging), let $\mathbf{A}^\dagger$ and $\mathbf{B}^\dagger$ denote analogous events which also include logging at node SQ, for example, $\mathbf{A}^\dagger = \langle \{A, C, SQ\}, d_A = d_C = d_{SQ} \rangle$. Finally, recall that $\star$ denotes any sequence of events, $(\bigvee_{N' \subseteq N, N' \neq \emptyset} \langle N', \mathsf{true} \rangle)^*$.

The following formula describes the alternation between events X and Y:

$$P(X, Y) = \mathsf{A}[[(X; Y)^*]] \langle\!\langle X \rangle\!\rangle \mathsf{true} \wedge \mathsf{A}[[(X; Y)^*; X]] \langle\!\langle Y \rangle\!\rangle \mathsf{true}.$$

In words, $P(X, Y)$ states that it is always possible to do an X after a series of $X;Y$s, and that it is always possible to do a Y after a series of $X;Y$s followed by an X. It can be shown that the connectors 2(a), 2(b) and 3 satisfy properties $P(\mathbf{A}, \mathbf{B})$, $P(\mathbf{A'}, \mathbf{B'})$, and $P(\mathbf{A}^\dagger, \mathbf{B}^\dagger)$, respectively.

[4] This is a valid step because it uses the present theorem inductively on ρ which is a smaller formula than $\mathsf{E}\rho$.

Adding Logging: Let F be the construction corresponding to adding logging (Example 3.2) and F^{-1} corresponding to removing logging (Example 3.3). Firstly, we'd like to reason that adding logging has no effect on the original operation of the connector. A formula stating part of this requirement is

$$\mathsf{A}[\![F]\!]P(\mathbf{A}^\dagger, \mathbf{B}^\dagger),$$

as the events $\mathbf{A}^\dagger$ and $\mathbf{B}^\dagger$ encompass events $\mathbf{A}$ and $\mathbf{B}$. (The logic requires that we name all nodes which appear synchronously in a transition. Another approach would be to project the behaviour represented in the constraint automata onto some set of nodes that we are interested in [19].)

Secondly, the formula

$$\mathsf{A}[[\star]][\![F]\!]P(\mathbf{A}^\dagger, \mathbf{B}^\dagger) \vee P(\mathbf{B}^\dagger, \mathbf{A}^\dagger)$$

states that in any state of the connector, that is, after any number of steps, reconfiguration puts the connector into a state where every bid and response is logged.

Finally, we'd like to reason that the construction F^{-1} returns the circuit to its original behaviour. Note that F^{-1} produces garbage in the circuit, so in part we are reasoning that the garbage has no effect. A formula capturing part of the desired property is

$$\mathsf{A}[\![F]\!][[(\mathbf{A}^\dagger; \mathbf{B}^\dagger)^*]][\![F^{-1}]\!]P(\mathbf{A}, \mathbf{B}).$$

(Or more generally, $\mathsf{A}[\![F]\!][[\star]][\![F^{-1}]\!]P(\mathbf{A}, \mathbf{B})$.) Note that behavioural equivalence would probably be a more appropriate technique for reasoning about this sort of property.

Changing Bidder: Let G denote the reconfiguration taking connector Figure 2(a) to Figure 2(b). When reasoning about reconfiguration, we use formulæ which describe the state of a system before the reconfiguration, and then describe the expected behaviour after reconfiguration. In the simplest of cases, we would like to say that reconfiguration in any state results in a certain behaviour. For example,

$$\mathsf{A}[[\star]][\![G]\!](\langle\!\langle \mathbf{A}' \rangle\!\rangle \mathsf{true} \wedge P(\mathbf{A}', \mathbf{B}')) \tag{1}$$

denotes that it is possible to perform reconfiguration step G in any state and then begin the protocol represented by $P(\mathbf{A}', \mathbf{B}')$. We may also want to state that reconfiguration enables the components connected to the initial connector to finish their protocol. Following the bid-response protocol, we require every $\mathbf{A}$ to have a matching $\mathbf{B}$. A formula capturing part of this property is

$$\mathsf{A}[[\star; \mathbf{A}]][\![G]\!]\langle\!\langle \mathbf{B} \rangle\!\rangle \mathsf{true}. \tag{2}$$

Both Formulæ (1) and (2) are invalid in the model given by the automata in Figure 2(a). Rehashing Section 1.2, performing the action $\mathbf{A}$ and then reconfiguring results in a state where performing $\mathbf{B}$ is not possible, because node B is no longer connected. Furthermore, performing $\mathbf{A}'$ is also impossible, as the connector is in a state only enabling $\mathbf{B}'$.

The following formula specifies when it is possible to reconfigure in a way that preserves both protocols:

$$\mathsf{A}[[(\mathbf{A}; \mathbf{B})^*]][\![G]\!](\langle\!\langle \mathbf{A}' \rangle\!\rangle \mathsf{true} \wedge P(\mathbf{A}', \mathbf{B}')).$$

This means that states in which it is safe to perform the reconfiguration are those which occur after a response has been received. Thus if the A-B bidder is in control of reconfiguration, it must ensure

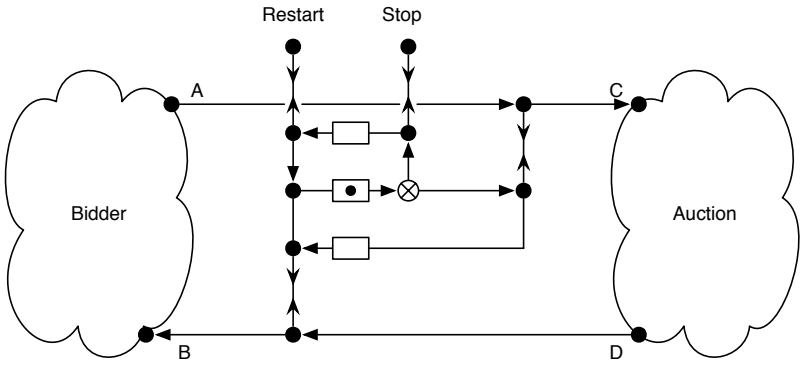


Figure 10. Connector facilitating safe external reconfiguration. Reconfiguration can occur between a *stop* and a *restart* action. A *stop* can never occur between an $\mathbf{A}$ and a $\mathbf{B}$ action. $\otimes$ denotes an exclusive router [2] which chooses which of the two loops of FIFO1 buffers receives the token. Each loop enforces part of the protocol.

that reconfiguration occurs after a $\mathbf{B}$. If however, the reconfiguration is done independently of the A-B bidder, the connector must be changed to give the reconfigurer a means for stopping the connector only after a response, in order to perform the reconfiguration, and then to restart the connector after reconfiguration. Figure 10 shows the required modification. This technique applies in many situations, but it could become a source of inefficiency if too much of a connector is stopped for too long in order to perform the reconfiguration.

Stock Price Watcher Consider the stock price watcher systems described in Figures 4 and 5. Let P_1, P_2, P_3, and P_4 represent the events corresponding to data emerging from the four stock price sources (P_1 is the original one). Similarly, let D_1, D_2, D_3, and D_4 represent that the respective stock prices are passed to the display component. Finally, let X denote the event corresponding to the publishing of an exchange rate. For simplicity, we ignore all other events apart from these.

We will assume that the convert component works *asynchronously*, meaning that its behaviour can be described by the schedule expression $(\mathit{price}; \mathit{rate} \vee \mathit{rate}; \mathit{price}); \mathit{result})^*$, where the events *price*, *rate*, and *result* correspond to some price or exchange rate being passed to the convert component, and to some result being produced, respectively.

The behaviour of the system in Figure 4(a) can be captured in part by the following formula:

$$\mathsf{A}\langle\!\langle((P_1; X \vee X; P_1); D_1)^*\rangle\!\rangle\mathsf{true}.$$

Now let F, G and H be the reconfiguration operations taking the initial connector (Figure 4(a)) to the second (Figure 4(b)), and the second to the third (Figure 5(a)), and the third to the fourth (Figure 5(b)), respectively.

We have, for instance, that each reconfiguration step never leads to a deadlocked connector, regardless of the state in which the reconfiguration occurs:

$$\mathsf{A}[[\star]][F][[\star]]\mathsf{Xtrue} \wedge \mathsf{A}[[\star]][F][[\star]][G][[\star]]\mathsf{Xtrue} \wedge \mathsf{A}[[\star]][F][[\star]][G][[\star]][H][[\star]]\mathsf{Xtrue}.$$

We can be more specific in that after any of these reconfigurations, it is possible within two steps to display one of the converted prices. Let S denote a single step not involving a display D_i, that is $S = P_1 \vee P_2 \vee P_3 \vee P_4 \vee X$. The formula representing this property for reconfiguration operation F is:

$$\mathsf{A}[[\star]][F][[\epsilon \vee S \vee S; S]]\mathsf{E}\langle\langle D_i \rangle\rangle \mathsf{true}.$$

Similar properties hold for the other reconfiguration operations, even when arbitrary steps occur between reconfiguration. Other similar properties can easily be expressed.

8. Related Work

Technology and techniques for reconfigurable systems come in different guises: mobile agents [33], dynamic rebinding of libraries [5], component-hot swapping, and via a coordination layer, whether it be a tuple space [25], a tool bus [11], or component connector [2, 36, 31]. Formalisms for reasoning about mobility in effect reason about reconfiguration, in the setting where the behaviour of the entitiy can depend upon its location. Examples include the ambient calculus and its logics [16], Klaim [13], the lambda calculus of dynamic rebinding [14], high-level Petri nets [40], and so on. The present work is the first we are aware of for reasoning about the reconfiguration of Reo software connectors.

Bruni *et al*. [15] provide a complete axiomatization of a simple language of connectors. Their axiomatization accounts for a single step of a Reo connector which does not consider the details of data, that is, it considers just synchronisation and exclusion. Issues such as data, buffers, and hence state, and reconfiguration are beyond the scope of their formalism.

Reo connectors have a *prima facie* resemblance to Petri nets [37], though the differences between the two are quite significant. Petri nets offer synchronisation only at each 'transition' of a net, whereas synchronisation can be more global in a Reo connector, dictated by the kinds of channels connected together. A number of attempts have been made to link Reo and Petri nets. The first attempt encoded MoCha, a predecessor to Reo, into Petri nets [30]. MoCha has the same primitive channels as Reo, but lacks the compositional synchronisation and exclusion of Reo, and is thus much simpler. At attempt to encode this aspect of Reo into a low-level Petri net proved to be too difficult, due to the more global synchronisation in Reo, though encoding the same Petri net model into Reo was trivial.[5] Choosing a more expressive kind of Petri net may lead to better results.

Some forms of Petri nets offer reconfiguration. Object nets, for example, deal with reconfiguration of systems by having, in essence, Petri nets as the tokens of other Petri nets [40]. These models have been used for modelling dynamic reconfiguration [26]. The result is a powerful, yet difficult to understand, model. Reconfiguration in Reo is, at least, simpler to understand. One significant difference is that these models perform reconfiguration within the Petri net, not from outside as in our model. Work is afoot to change this in the setting of Reo, though, to a large extent, reconfiguration must at least be initiated from outside of a connector, since all reconfiguration scenarios cannot be known at the time a connector is deployed.

There are a number of approaches to reasoning about Petri nets, some of which are applicable to reconfiguration. The first approach involves using CTL*, much as we have done here, though the step from a Petri net to a model for the logic is simpler than our path through constraint automata, for simple Petri

[5] This work is due to Juan Guillen-Scholten, though he never wrote it up.

net models, though significantly more complicated for various coloured Petri nets [29]. As reconfiguration is integrated into the advanced Petri nets models themselves, an additional reconfiguration modality is not needed to reason about them, though on-the-fly model checking techniques are. Another approach is to use an algebraic semantics of Petri nets in terms of rewriting logic [39]. Formulæ in this formalism can the be verified using Maude's toolset [22]. This approach is very general and may be fruitfully adapted to reasoning about Reo, though we would first need to perform various encodings and argue for their correctness. The approach presented in this paper avoids this by building heavily on existing work. Approaches based on process algebra exist and can use the variety of tools available for reasoning about them [29]. Again the difficulty here is the effort required to develop a process algebraic model of Reo connectors that has more than simply local synchronisation, and then to adapt all the logical tools to the resulting process algebra. A final alternative converts Petri nets into linear logic formulæ and reasons about using tools for linear logic [27]. The problem is that such tools are not so well-developed as tools for model checking CTL*.

A number of approaches to reasoning about and model checking pointer programs with dynamically changing heaps exist [24, 43]. These differ from the work presented here in that they focus on the underlying intensional structure of the heap, whereas our focus is on connector behaviour. The result is that our logic need not keep track of particularly complicated structures such as heaps.

Interestingly, logics such as the Logic of Public Announcements [8] and Sabotage Logic [10] also include modalities for jumping between models. Neither logic is based on CTL*, so they are not readily comparable to ReCTL*. Further afield, Verbaan *et al*. [41] model evolving systems in terms of lineages of automata in order to study non-uniform complexity theory. A jump between automata in their model is spontaneous, whereas ours result from a specific construction. No logical tools are provided for reasoning about their automata.

9. Conclusion and Future Work

We presented the semantics of Reo connectors in the presence of reconfiguration, a logic for reasoning about the reconfiguration of running connectors, and a model checking algorithm for the logic. We also indicated problems that may occur when reconfiguring a connector which enforces a software protocol, and gave one way of overcoming such problems.

As an anonymous referee correctly observed, the reconfiguration operations require that the semantics of a connector be recomputed from scratch. This can be optimised by caching, which would reduce the complexity of model checking, trading time for space. However, this fact suggests that a better choice of reconfiguration operations is required. Admittedly, the reconfiguration operations are rather low level, but they are universal in that any other reconfiguration operations can be defined in terms of them. We think that the problem is not with the reconfiguration operations, per se, but due to the fact that in automata-based models performing reconfiguration on an automata generally requires that a new automata be totally computed before model checking can occur. Thus, we need to bear the cost, as we are working with an automata-based model. Reconfiguration occurs seamlessly in process algebra or labelled transition systems, because semantics in such set-ups are computed on demand. Recasting our semantics in this direction, or performing on-the-fly model checking [9], might reduce the costs due to reconfiguration.

Directions for future work include adding components and Reo's *connect* and *disconnect* opera-

tions [2] to the model. Adding components may seem strange, given that we are working in an exogenous coordination setting which has the advantage of being able to abstract from components. There are two reasons why this would be fruitful. Firstly, in order to establish the correctness of the entire system, one needs to consider both the components (or an appropriate abstraction of them, perhaps in terms of constraint automata) and the connector connecting them. Secondly, it would be interesting to investigate the degree to which Reo can perform software adaptation [45]. Being able to exogenously adapt components would reduce the need for replacing components when performing reconfiguration. We would also like to find more convenient and automatic ways for reasoning about the interaction between protocols and reconfiguration and for repairing problems that may arise. Finally, exploring counterexample generation will make our model checking algorithm more useful in this respect. A good starting point is the article of Clarke *et al*. [20].

The model of reconfiguration in Reo presented in this paper was stratified in that reconfiguration could modify the connector, but could not depend upon the connector's behaviour. This stratification justified our approach to dealing only with reconfigurations which were essentially straight-line programs. However, work is presently being done to determine the semantics of Reo when data flow can influence reconfiguration. Indeed, the idea has been posited to embed reconfiguration operations into the connectors themselves, but, as yet, no conclusive results have been obtained, due to the unsatisfying interaction between synchronisation and reconfiguration. Should this idea come to fruition, the logic presented in this paper would need a significant overhaul. Nevertheless, the approach taken here is adequate for the existing Reo semantics.

We leave open the question of whether the reconfiguration modality can be encoded in a logic without it. One possible attack is to delve into the model checker dSPIN [23], which is an extension of the SPIN model checker that can deal with dynamic object structures and object creation. Finally, we also would like to explore meta-theoretic properties of our logic. For example, what is the equivalence induced by ReCTL*? We suspect that the result will be rather disappointing, as reconfiguration can arbitrarily change any visible part of a connector, and thus can be used to reveal the differences between two different connectors that may be behaviourally equivalent before reconfiguration.

Acknowledgements Many thanks goes to the many hands, anonymous and otherwise, this document has passed through on the way to making it what it is today.

References

[1] Abdulla, P. A., Baier, C., Iyer, S. P., Jonsson, B.: Reasoning about Probabilistic Lossy Channel Systems, *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, Springer-Verlag, London, UK, 2000, ISBN 3-540-67897-2.

[2] Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition, *Mathematical Structures in Computer Science*, **14**(3), June 2004, 329–366.

[3] Arbab, F.: Abstract Behavior Types: a foundation model for components and their composition, *Science of Computer Programming*, **55**, 2005, 3–52.

[4] Arbab, F., Baier, C., de Boer, F., Rutten, J.: Models and Temporal Logics for Timed Component Connectors, *IEEE International Conference on Software Engineering and Formal Methods (SEFM '04)*, IEEE Computer Society, Beijing, China, September 2004, ISBN 0-7695-2222-X.

[5] Armstrong, J., Williams, M., Wikstrom, C., Virding, R.: *Concurrent Programming in Erlang*, Prentice-Hall, New Jersey, USA, 1996.

[6] Arndol, A.: *Finite Transition Systems*, Prentice Hall, 1994.

[7] Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata, *Science of Computer Programming*, **61**(2), 2006, 75–113, ISSN 0167-6423.

[8] Baltag, A., Moss, L. S., Solecki, S.: A logic of public announcements, common knowledge, and private suspicions, *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK-98), Evanston, IL, USA* (I. Gilboa, Ed.), Morgan Kaufmann, 1998.

[9] Beer, I., Ben-David, S., Landver, A.: On-the-Fly Model Checking of RCTL Formulas, *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, number 1427 in Lecture Notes in Computer Science, Springer-Verlag, London, UK, 1998, ISBN 3-540-64608-6.

[10] van Bentham, J.: An essay on sabotage and obstruction, *Festschrift in Honour of Jörg Siekman* (D. Hutter, S. Werner, Eds.), number 2605 in Lecture Notes in Computer Science, 2005.

[11] Bergstra, J., Klint, P.: The discrete time ToolBus – A software coordination architecture, *Science of Computer Programming*, **31**(2-3), July 1998, 205–229.

[12] Berry, G.: The Foundations of Esterel, in: *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, 2000, 425–454.

[13] Bettini, L., Nicola, R. D., Loreti, M.: Formalizing Properties of Mobile Agent Systems, *Coordination Languages and Models*, number 2315 in Lecture Notes in Computer Science, 2002.

[14] Bierman, G., Hicks, M., Sewell, P., Stoyle, G., Wansbrough, K.: Dynamic rebinding for marshalling and update, with destruct-time λ, *International Conference on Functional Programming (ICFP)*, August 2003.

[15] Bruni, R., Lanese, I., Montanari, U.: A Basic Algebra of Stateless Connectors, *Theoretical Computer Science*, **366**(1–2), November 2006, 98–120.

[16] Cardelli, L., Gordon, A. D.: Ambient Logic, *Mathematical Structures in Computer Science*, 2005, To Appear.

[17] Clarke, D. G.: *Reasoning about connector reconfiguration I: Equivalence of constructions*, Technical Report SEN-R0506 ISSN 1386-369X, CWI, Amsterdam, The Netherlands, 2005.

[18] Clarke, D. G.: Reasoning about Connector Reconfiguration II: Basic Reconfiguration Logic, *IPM International Workshop on Foundations of Software Engineering (FSEN'05)* (F. Arbab, M. Sirjani, Eds.), number 159 in ENTCS, Elsevier, 2005.

[19] Clarke, D. G., Costa, D., Arbab, F.: Modelling Coordination in Biological Systems, *Int. Symposium on Leveraging Applications of Formal Methods (ISoLA 2004)*, number 4313 in Lecture Notes in Computer Science, Springer-Verlag GmbH, October 2006.

[20] Clarke, E., Jha, S., Lu, Y., Veith, H.: Tree-Like Counterexamples in Model Checking, *Logic in Computer Science*, 2002, 19–26, ISSN 1043-6871.

[21] Clarke Jr., E. M., Grumberg, O., Peled, D. A.: *Model Checking*, MIT Press, 1999.

[22] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J. F.: Maude: Specification and Programming in Rewriting Logic, *Theoretical Computer Science*, **285**, August 2002, 187–243.

[23] Demartini, C., Iosif, R., Sisto, R.: dSPIN: A Dynamic Extension of SPIN, *Theoretical and Practical Aspects of SPIN Model Checking* (D. Dams, R. Gerth, S. Leue, M. Massnik, Eds.), number 1680 in Lecture Notes in Computer Science, Springer, 1999.

[24] Distefano, D.: *On model checking the dynamics of object-based software: a foundational approach*, Ph.D. Thesis, University of Twente, Nov 2003.

[25] Ducasse, S., Hofmann, T., Nierstrasz, O.: OpenSpaces: An object-oriented framework for reconfigurable coordination spaces, *Coordination Languages and Models*, number 1906 in Lecture Notes in Computer Science, September 2000.

[26] Farwer, B.: Dynamic modification of object Petri nets: an application to modelling protocols with fork-join structures, *Fundamenta Informaticae*, **51**(1), 2002, 91–101, ISSN 0169-2968.

[27] Farwer, B., Misra, K.: Dynamic Modification of System Structures Using LLPNs, *Proceedings of Andrei Ershov Fifth International Conference* PERSPECTIVES OF SYSTEM INFORMATICS, number 2890 in Lecture Notes in Computer Science, July 2003.

[28] Ghassemi, F., Tasharofi, S., Sirjani, M.: Automated mapping of Reo circuits to constraint automata, *IPM International Workshop on Foundations of Software Engineering (FSEN'05)* (F. Arbab, M. Sirjani, Eds.), number 159 in ENTCS, Elsevier, 2005.

[29] Girault, C., Valk, R., Eds.: *Petri Nets for Systems Engineering*, Springer-Verlag, 2003.

[30] Guillen-Scholten, J., Arbab, F., de Boer, F. S., Bonsangue, M. M.: Modeling the Exogenous Coordination of Mobile Channel-based Systems with Petri Nets., *FOCLASA*, number 154 in ENTCS, 2006.

[31] Hirsch, D., Inveradi, P., Montanari, U.: Reconfiguration of Software Architecture Styles with Name Mobility, *Coordination Languages and Models*, number 1906 in Lecture Notes in Computer Science, September 2000.

[32] Holzmann, G. J.: *The SPIN Model Checker*, Addison Wesley, 2003.

[33] Milner, R.: *Communicating and Mobile Systems: the Pi-Calculus*, Springer Verlag, 1999.

[34] Nicola, R. D., Fantechi, A., Gnesi, S., Ristori, G.: An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems., *Computer Networks and ISDN Systems*, **25**(7), 1993, 761–778.

[35] Papadopoulos, G. A., Arbab, F.: Coordination models and languages, *M. Zelkowitz (Ed.), The Engineering of Large Systems*, 46, Academic Press, 1998.

[36] Papadopoulos, G. A., Arbab, F.: Configuration and dynamic reconfiguration of components using the coordination paradigm, *Future Generation Computer Systems*, **17**, 2001, 1023–1038.

[37] Petri, C. A.: *Kommunikation mit Automaten*, Ph.D. Thesis, University of Bonn, 1962.

[38] Rabinovich, A.: Quantitative analysis of probabilistic lossy channel systems, *Inf. Comput.*, **204**(5), 2006, 713–740, ISSN 0890-5401.

[39] Stehr, M.-O., Meseguer, J., Ölveczky, P. C.: Rewriting Logic as a Unifying Framework for Petri Nets, *Unifying Petri Nets (Advances in Petri Nets)* (H. Ehrig, G. Juhas, J. Padberg, G. Rozenberg, Eds.), number 2128 in Lecture Notes in Computer Science, Springer-Verlag, 2001.

[40] Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets, *ICATPN '98: Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, number 1420 in Lecture Notes in Computer Science, Springer-Verlag, London, UK, 1998, ISBN 3-540-64677-9.

[41] Verbaan, P. R. A., van Leeuwen, J., Wiedermann, J.: *Lineages of automata*, Technical Report UU-2004-018, Utrecht University: Information and Computing Sciences, 2004.

[42] Wong, S. K. M., Wang, L. S., Yao, Y. Y.: Interval structure: a framework for representing uncertain information, *Proceedings of the Eighth conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992, ISBN 1-55860-258-5.

[43] Yahav, E., Reps, T. W., Sagiv, S., Wilhelm, R.: Verifying Temporal Heap Properties Specified via Evolution Logic., *ESOP* (P. Degano, Ed.), number 2618 in Lecture Notes in Computer Science, Springer, 2003.

[44] Yao, Y. Y., Lin, T. Y.: Generalization of rough sets using modal logic, *Intelligent Automation and Soft Computing*, **2**, 1996, 103–120.

[45] Yellin, D. M., Strom, R. E.: Protocol specifications and component adaptors, *ACM Trans. Program. Lang. Syst.*, **19**(2), 1997, 292–333, ISSN 0164-0925.

[46] Zlatev, Z., Diakov, N., Pokraev, S.: Construction of Negotiation Protocols for E-Commerce Applications, *ACM SIGecom Exchanges*, **5**(2), November 2004, 12–22.