Software ENgineering

Reasoning about connector reconfiguration II: basic reconfiguration logic

D.G. Clarke

REPORT SEN-R0607 MAY 2006

ISSN 1386-369X

Reasoning about connector reconfiguration II: basic reconfiguration logic

ABSTRACT
Software systems evolve over time. To facilitate this, the coordination language Reo offers operations to dynamically reconfigure the topology of component connectors. We present a semantics of Reo in the presence of reconfiguration, and a logic, and its model checking algorithm, for reasoning about connector behaviour in this setting.

2000 Mathematics Subject Classification: none
1998 ACM Computing Classification System: D.2.4, I.2.2, F.1.1
Keywords and Phrases: Dynamic Reconfiguration, Coordination, Component Connectors, Reo
Note: This work was carried out under project SEN3 - CQuattro.

Reasoning about Connector Reconfiguration II: Basic Reconfiguration Logic

Dave Clarke
CWI
Amsterdam, Netherlands
dave@cwi.nl

May 15, 2006

Abstract

Software systems evolve over time. To facilitate this, the coordination language Reo offers operations to dynamically reconfigure the topology of component connectors. We present a semantics of Reo in the presence of reconfiguration, and a logic, and its model checking algorithm, for reasoning about connector behaviour in this setting.

1 Introduction

Software systems evolve over time. Continuously running distributed systems, in particular, require extensive support to facilitate evolution, deployment, upgrading, and reconfiguration. Coordination languages offer a technology to address this issue, providing the glue for plugging software components together. The channel-based coordination language Reo [1] provides *connectors* for connecting software components in such a way that the components are unaware of their role in the composed software. Reo's connectors resemble electronic circuits, where channels corresponding to wires and electronic components are connected at "nodes", and software components are connected at the boundary of the connector. Reconfiguration in this setting corresponds to changing the connector between components.

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. To this end, this paper makes the following *contributions*: 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.

1.1 Overview of Reo

Reo is a channel-based coordination language based on circuit-like *connectors* which coordinate software components. (For a detailed account, see Arbab [1].) Various kinds of channel are possible, offering different synchronisation, buffering, lossy and even directionality policies. Each channel imposes synchronisation or exclusion constraints on dataflow through its ends. If we consider that an event corresponds to the flow of data through a channel end, the behaviour of a connector at any given time step is to permit some collection of the possible events to occur whilst excluding the possibilitiy of others. The *synchronisation* of two events means that they will either both occur or both not occur in a particular step. *Exclusion* of two events means that both cannot occur in a particular step. Channels are connected at *nodes* which 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 [1, 2].

An example connector is shown in Figure 1(a). This connector uses three channel types. A *synchronous channel* (ordinary arrow, such as A-a) sends data from one end to the other, synchronously. An empty *FIFO buffer of size one* (arrow with box, i-j), dubbed FIFO1, allows a write to its input end (i) to succeed, filling the buffer, and a full buffer (box containing data, h-g) allows a take from its output end (g) to succeed, emptying the buffer. A *synchronous drain* (arrow heads pointing inwards, e-f) requires that two writes to its ends occur synchronously, and the data is lost. The connector in the figure first allows 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 FIFO1 buffers sequences these two events.

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

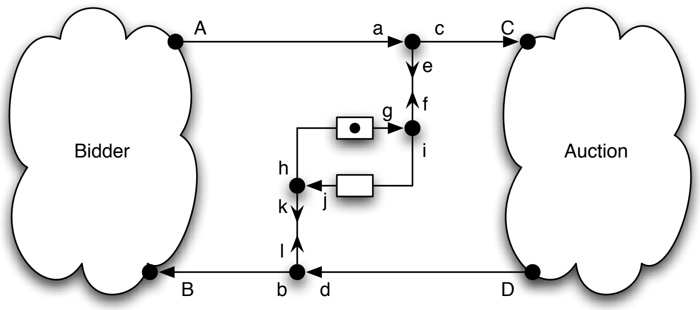
Consider a distributed system with two kinds of components: one managing an auction and one issuing an individual bidder's bids. Bids are routed via a Reo connector, see Figure 1(a), to an auction component, which then issues a response indicating the outcome of the bid. The Reo connector guarantees that the simple protocol, alternating bids and responses, is preserved. This scenario has been adapted from the application of Reo to auction protocols [22].

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

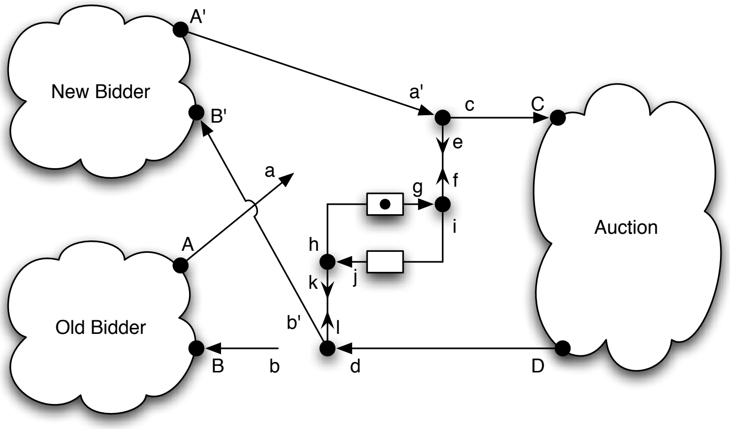
- new bidders join an auction and their components are connected;
- bidders leave an auction and their components are disconnected;
- an auction or bidder component is upgraded and replaced;
- the underlying bid-response protocol, enforced by the Reo connector, is modified, for example, to include an 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 wish to log all bids and their corresponding responses. To do so requires the addition of the channels highlighted in reconfigured connector in Figure 2, with the Logging component attached at node SQ. Subsequently, we may remove the logging. We wish to reason that (1) logging does not affect the bid-response protocol; (2) removal of the logging mechanism produces a connector with the same behaviour as the original.



(a)



(b)

Figure 1: (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'.

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 1(b) with channels A-a and B-b disconnected; and finally, the response corresponding to the bid is received at node B', which may not be expecting it, instead of at 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.

We revisit these scenarios in Section 7.

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

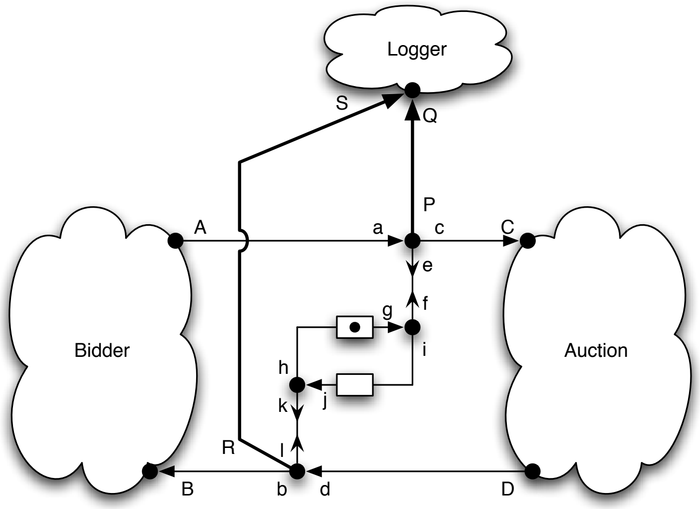


Figure 2: Logging Component added at SQ. Bids and responses are copied to SQ.

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*).[1] 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. 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, this 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. *$B \cup H$ is a partition of the set 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 2.2 *The connector in Figure 1(a) is represented by*

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

[1]This terminology differs from Arbab [1] who uses the phrases *source* for *input* and *sink* for *output*. Read *input* as *accepting input* and *output* as *producing output*.

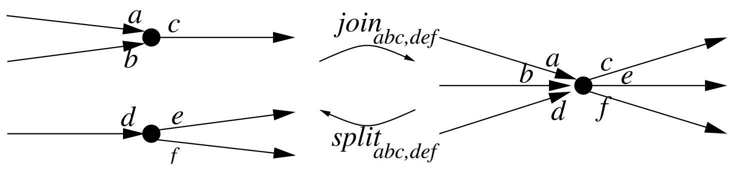


Figure 3: Joining, $\texttt{join}_{abc,def}$, groups nodes abc and def together to form node $abdcef$. Splitting, $\texttt{split}_{abc,def}$, performs the inverse operation. All possible ways of splitting and joining are permitted. Both operations, however, tend to drastically alter data flow.

3 Constructions on Reo Connectors

Reo has a language for constructing and reconfiguring connectors [1], the essence of which is captured in the following language of *constructions*:

$$G \quad ::= \quad \texttt{id} \mid GG \mid Ch^T_{\mathsf{a},\mathsf{b}} \mid \texttt{join}_{a,b} \mid \texttt{split}_{a,b} \mid \texttt{hide}_a \mid \texttt{forget}_a$$

Constructions ($\mathcal{C}on$) are (partial) operations taking a Reo connector to another Reo connector (thus its 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 [12] 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:*

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

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

The identity construction, `id`, does not modify its argument. Sequential composition of F followed by G, is denoted GF, following the mathematical convention. Construction $Ch^T_{\mathsf{a},\mathsf{b}}$ 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 3). Construction $\texttt{hide}_a$ makes a mixed node operate 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 [1]. 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.

Example 3.2 *The connector in Figure 1(a) is a result of the construction:*

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

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

Example 3.3 *The reconfiguration producing the connector in Figure 2 from the one in Figure 1(a) is:*

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

Example 3.4 *A construction which takes the connector in Figure 2 and reproduces the connector in Figure 1(a), with some garbage, is:*

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

The actual Reo control language [1] is more involved as it is embedded in a programming language and is more convenient to use. We have adopted a simplified and clean core which takes a bird's eye view of reconfiguration in order to obtain the results presented here.

4 Constraint Automata: A Semantics for Reo

Constraint automata describe the data flow through nodes and the synchronisation and exclusion constraints on nodes in a Reo connector [4]. 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 $N \subseteq B$.

Definition 4.1 (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 [4] also included the set of initial states. Our presentation has slightly different requirements, so we've removed them and introduced a function *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 in Section 5. We have also added the notion of *trivial loop* which similifies the definition of product.

Arbab *et al* [4] 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 4.2 *The constraint automaton in Figure 4(a) models the connector in Figure 1(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 4(b) is a constraint automaton modelling the behaviour resulting from adding logging at node SQ (cf. Figure 2). The transitions indicate that logging occurs synchronously with both bids and responses, copying the data in both cases.*

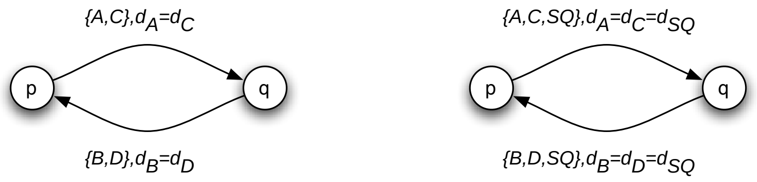


Figure 4: Example Constraint Automata. Trivial loops omitted.

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 nodes 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 [4]) *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 $\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, B_1 \cup B_2, \longrightarrow)$, 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 [4]) *Given constraint automaton $\mathcal{A} = (Q, B \uplus \{a\}, \longrightarrow)$, the automaton $\mathtt{hide}_a(\mathcal{A}) = (Q, B, \longrightarrow_a)$, 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 $q \xrightarrow{N',g'}_a r$.*

Consult Arbab *et al* [4] 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 $\mathtt{forget}_a$ 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, $\mathtt{forget}_a(\mathcal{A})$, 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 4.6 *If $\mathtt{forget}_A$ is applied to the constraint automaton in Figure 4(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.*

A note on garbage In the prequel [12] we determined when certain parts of a circuit corresponded to garbage and we claimed that removing the garbage caused no problem behaviourally. We give this result here as Theorem 4.7. 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 4.7 *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 trace equivalence as defined by Arbab *et al* [4]. 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 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. We anticipate that the results presented in this section also hold for a suitable notion of bisimilarity.

5 Reconfiguration Logic — ReCTL*

This section presents the logic ReCTL* for reasoning about reconfiguration. ReCTL* combines the well-known CTL* [13] with TSDSL (timed scheduled-data-stream logic) [3][2] 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 notions of *data constraint satisfaction* and *schedule expression*.

Definition 5.1 (Data Constraint Satisfaction) *Satisfaction of a data constraint g by a data assignment $\delta : Node \rightharpoonup_{\text{fin}} Data$ is denoted $\delta \models g$ and defined:*

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

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 [3]:

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

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 5.2 *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.*

[2]An anonymous referee pointed out that RCTL [7] 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.

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 $\mathsf{E}-$ and $- \,\mathsf{U}\, -$ are standard from CTL* [13]. 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 [3]. 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 satisfy 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.

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. 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.

5.1 Semantics of ReCTL*

The semantics of Reo in the presence of reconfiguration forms the basis of the semantics of ReCTL* formulæ. 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 approapriate constraint automaton (Definition 5.3); and secondly, a reconfiguration step can be performed — this is modelled as a reconfiguration transition *between* automata (Definition 5.7). 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 5.3 (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 5.4 (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 π_j be the sequence of labels of the prefix of π preceding the jth element, defined as:

$$\pi_0 \mathrel{\widehat{=}} \epsilon \qquad\qquad (q \xrightarrow{\delta} \pi)_{j+1} \mathrel{\widehat{=}} \delta.\pi_j.$$

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 $\mathtt{hide}_a$ takes behaviour into account.

Definition 5.5 (State Transfer) *The state transfer function for applying construction G to connector $\mathcal{C}$, $\mathcal{S}_{G,\mathcal{C}}(-) : State_{\mathcal{C}} \to \mathcal{P}(State_{G(\mathcal{C})})$, is defined:*

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

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

Lifting to sets of states, we obtain:

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

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

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

Lemma 5.6 $\mathcal{S}_{F,\mathcal{C}}(State_{\mathcal{C}}) = State_{F(\mathcal{C})}$.

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, $\mathcal{A} = \mathcal{R}[\![\mathcal{C}]\!] = \langle Q, B, \longrightarrow\rangle$ 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 5.7 (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) $\mathcal{A}' = \mathcal{R}[\![G(\mathcal{C})]\!] = (Q', B', \longrightarrow')$, (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 $\mathcal{S}_{G,\mathcal{C}}(V)(a_0) = \mathcal{S}_{G,\mathcal{C}}(V(a_0))$.

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 5. 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

The model checking algorithm for ReCTL* derives from those of CTL* [13] and TSDSL [3]. 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*. The model checking question is, given *connector* $\mathcal{C}$, which states of its underlying automaton satisfy state formula ψ:

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

where $State(\mathcal{A})$ is the set of states of the constraint automaton $\mathcal{A}$.

$$
\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\!|G|\!\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_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 5: Semantics of ReCTL*.

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 [13]. This fragment is the $\mathsf{E}-$ and $\langle\!|G|\!\rangle-$ free fragment of ReCTL*, which corresponds to the time-free fragment of TSDSL [3]:

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

The model checking algorithm for TSDSL [3] can be adapted to compute the following:

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

This algorithm clearly satisfies the following property:

Lemma 6.1 ([3]) $\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \mathsf{A}\rho^\dagger$ *if and only if* $q \in \mathcal{MC}_{TSDSL}(\mathcal{C}, \mathcal{A}, V, \mathsf{A}\rho^\dagger)$.

6.2 Model Checking ReCTL*

Figure 6 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 5.5). 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

$$\mathcal{S}^{\cap}_{G,\mathcal{C}}(Q) \ : \ \mathcal{P}(\mathit{State}_{G(\mathcal{C})}) \rightarrow \mathcal{P}(\mathit{State}_{\mathcal{C}}) = \{q \in \mathit{State}_{\mathcal{C}} \mid \mathcal{S}_{G,\mathcal{C}}(q) \cap Q \neq \emptyset\}$$

is then applied to the resulting state set to take 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.

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

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

Figure 6: 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.

The standard technique for checking formulæ of the form $\mathsf{E}\rho$ is used [13]. 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 $\mathcal{E}\mathit{lim}(-)$ performs the desired operation, mapping an ReCTL* formula to a TSDSL formula and valuation, which are then fed into $\mathcal{MC}_{TSDSL}(-)$.

6.3 Properties

The graph of constraint automata which 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 need be explored. On-the-fly model checking deals with infinite state models in a similar manner as we do [18]. Model checking within a given constraint automaton is bounded (though Arbab *et al* [3] present no complexity results we can drawn from), and thus our model checking algorithm is also bounded. We anticipate 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 [16], 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 for all $Q\subseteq\mathit{States}(\mathcal{A})$, $\langle\mathcal{C},\mathcal{A},V\cup\{a_0\mapsto Q\},\pi\rangle\models\rho$.*

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

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.
Proof of part 1, by case analysis on ψ.

1. Case true: immediate.

2. Case a_0: immediate.

3. 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 $\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \psi_2$ if and only if, by induction hypothesis, $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi_1)$ and $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi_2)$ if and only if $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi_1) \cap \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi_2)$ if and only if $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi_1 \wedge \psi_2)$.

4. Case $\neg\psi$: $\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \neg\psi$ if and only if $\langle \mathcal{C}, \mathcal{A}, V, q\rangle \not\models \psi$ if and only if, by induction hypothesis, $q \notin \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \psi)$ if and only if $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \neg\psi)$.

5. 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 if and only if, by part 2 of this lemma, $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi\rangle \models \rho'$, where $\mathcal{E}lim(\rho) = (V', \rho')$. 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 $\langle\!\langle G\rangle\!\rangle-$ free, this is, by Lemma 6.1, equivalent to $q \in \mathcal{MC}_{TSDSL}(\mathcal{C}, \mathcal{A}, V \cup V', \mathsf{E}\rho')$, which is equivalent to $q \notin \mathcal{MC}_{TSDSL}(\mathcal{C}, \mathcal{A}, V \cup V', \mathsf{A}\neg\rho')$, which is in turn equivalent to $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \mathsf{E}\rho)$.

6. Case $\langle\!\langle G\rangle\!\rangle\psi$: $\langle \mathcal{C}, \mathcal{A}, V, q\rangle \models \langle\!\langle G\rangle\!\rangle\psi$ if and only if, by Definition 5.7, 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 $q' \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}', \mathcal{A}', V', \psi)$. Setting $Q == \mathcal{MC}_{ReCTL^*}(\mathcal{C}', \mathcal{A}', V', \psi)$, 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 $q' \in \mathcal{S}^{\cap}_{G,\mathcal{C}}(\mathcal{MC}_{ReCTL^*}(\mathcal{C}', \mathcal{A}', V', \psi))$, and hence $q' \in \mathcal{MC}_{ReCTL^*}(G(\mathcal{C}), \mathcal{C}, \mathcal{A}, V, \langle\!\langle G\rangle\!\rangle\psi)$.

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. Case true: immediate.

2. Case a_0: immediate.

3. 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 $\langle \mathcal{C}, \mathcal{A}, V, \pi\rangle \models \rho_2$ if and only if, by induction hypothesis, $\langle \mathcal{C}, \mathcal{A}, V \cup V_1, \pi\rangle \models \rho_1'$ and $\langle \mathcal{C}, \mathcal{A}, V \cup V_2, \pi\rangle \models \rho_2'$, where $(\rho_1', V_1) = \mathcal{E}lim(\rho_1)$ and $(\rho_2', V_2) = \mathcal{E}lim(\rho_2)$. 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 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 $(\rho_1' \wedge \rho_2', V_1 \cup V_2) = \mathcal{E}lim(\rho_1 \wedge \rho_2)$.

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

5. 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 if and only if, by part 1 of this lemma, $q \in \mathcal{MC}_{ReCTL^*}(\mathcal{C}, \mathcal{A}, V, \mathsf{E}\rho)$.[3] 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.

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

7. 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 $\pi_i \in \mathcal{L}(\alpha)$ 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 $\langle \mathcal{C}, \mathcal{A}, V \cup V', \pi\rangle \models \langle\!\langle\alpha\rangle\!\rangle\rho'$.

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

8. 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 $(\rho'_1 \mathsf{U} \rho'_2, V_1 \cup V_2) = \mathcal{E}lim(\rho_1 \mathsf{U} \rho_2)$.

7 Reconfiguration Scenarios, Revisited

ReCTL* can be used to describe the behaviour of a fixed connector as in Arbab *et al* [3]. For Figure 1(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 1(b) (changing bidders), let $\mathbf{A}'$ and $\mathbf{B}'$ denote analogous events in the reconfigured connector. For Figure 2 (adding logging), let $\mathbf{A}^\dagger$ and $\mathbf{B}^\dagger$ denote analogous events which also include logging at node SQ, for example, $\mathbf{A}^\dagger = \langle\{A, C, SQ\}, d_A = d_C = d_{SQ}\rangle$.

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 it always possible to do a Y after a series of $X;Y$s followed by an X. It can be shown that the connectors 1(a), 1(b) and 2 satisfy properties $P(\mathbf{A}, \mathbf{B})$, $P(\mathbf{A}', \mathbf{B}')$, and $P(\mathbf{A}^\dagger, \mathbf{B}^\dagger)$, respectively.

Adding Logging: Let F be the construction corresponding to adding logging (Example 3.3) and F^{-1} corresponding to removing logging (Example 3.4). 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 $\mathbf{B}$.

Secondly, we'd like to reason that the construction F^{-1} returns the circuit to its original behaviour. Not 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

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

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 1(a) to Figure 1(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, if we use $\star$ to denote any sequence of events, then

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

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 wish 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. 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

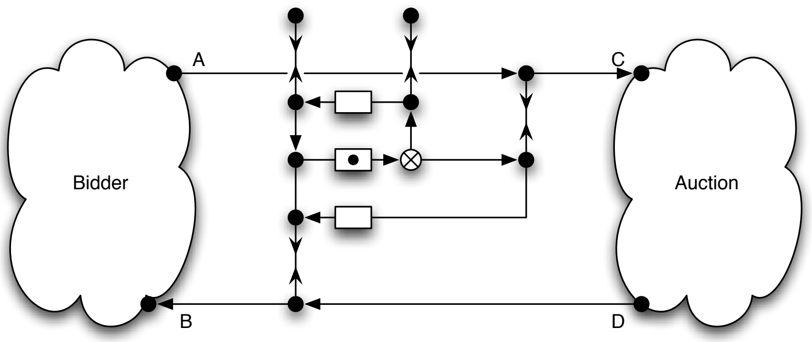


Figure 7: Connector facilitating safe external reconfiguration. Reconfiguration can occur between a *stop* and a *restart* action. A *stop* can never occur between an **A** and a **B** action. $\otimes$ denotes an exclusive router [1] which chooses which of the two loops of FIFO1 buffers receives the token. Each loop enforces part of the protocol.

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 that reconfiguration occurs after a **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 7 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.

8 Related Work

Technology and techniques for reconfigurable systems come in different guises: mobile agents, dynamic rebinding of libraries [5], component-hot swapping, and via a coordination layer, whether it be a tuple space [15], a tool bus [8], or component connector [1, 19, 17]. 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 [11], Klaim [9], the lambda calculus of dynamic rebinding [10], and so on. The present work is the first we are aware of for reasoning about the reconfiguration of Reo software connectors.

Interestingly, logics such as the Logic of Public Announcements [6] and Sabotage Logic [20] 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.* [21] 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 which occur when reconfiguring a connector that enforces a software protocol, and gave one way of overcoming such problems.

Directions for future work include adding components and Reo's connect and disconnect operations [1] to the model, and finding more convenient and automatic ways for reasoning about the interaction between protocols and reconfiguration and for repairing problems that may arise. 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 [14], which is an extension of the SPIN model checker that can deal with dynamic object structures and object creation. Finally, we also wish 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 connectors which may be equivalent before reconfiguration.

References

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

[2] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming*, 55:3–52, 2005.

[3] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logics for timed component connectors. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM '04)*, Beijing, China, September 2004.

[4] Farhad Arbab, Christel Baier, Jan Rutten, and Marjan Sirjani. Modeling component connectors in Reo by constraint automata. In *International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA)*, ENTCS, Marseille, France, September 2003. Elsevier Science.

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

[6] Alexandru Baltag, Lawrence S. Moss, and Sławomir Solecki. A logic of public announcements, common knowledge, and private suspicions. In Itzhak Gilboa, editor, *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK-98), Evanston, IL, USA*. Morgan Kaufmann, 1998.

[7] Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of rctl formulas. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 184–194, London, UK, 1998. Springer-Verlag.

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

[9] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Formalizing properties of mobile agent systems. In *Coordination Languages and Models*, volume 2315 of *LNCS*, pages 72–87, 2002.

[10] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *International Conference onf Functional Programming (ICFP)*, pages 99–110, August 2003.

[11] Luca Cardelli and Andrew D. Gordon. Ambient logic. *Mathematical Structures in Computer Science*, 2005. To Appear.

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

[13] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[14] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massnik, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 261–276. Springer, 1999.

[15] Stéphane Ducasse, Thomas Hofmann, and Oscar Nierstrasz. OpenSpaces: An object-oriented framework for reconfigurable coordination spaces. In *Coordination Languages and Models*, volume 1906 of *LNCS*, pages 1–18, September 2000.

[16] Fatemeh Ghassemi, Samira Tasharofi, and Marjan Sirjani. Automated mapping of Reo circuits to constraint automata. In Farhad Arbab and Marjan Sirjani, editors, *IPM International Workshop on Foundations of Software Engineering (FSEN'05)*, volume 159 of *ENTCS*. Elsevier, 2005.

[17] Dan Hirsch, Paola Inveradi, and Ugo Montanari. Reconfiguration of software architecture styles with name mobility. In *Coordination Languages and Models*, volume 1906 of *LNCS*, September 2000.

[18] Gerard J. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2003.

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

[20] Johan van Bentham. An essay on sabotage and obstruction. In D. Hutter and S. Werner, editors, *Festschrift in Honour of Jörg Siekman*, LNAI, 2002.

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

[22] Z. Zlatev, N. Diakov, and S. Pokraev. Construction of negotiation protocols for E-commerce applications. *ACM SIGecom Exchanges*, 5(2):12–22, November 2004.