Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Reasoning about connector reconfiguration I: Equivalence of constructions

D.G. Clarke

REPORT SEN-R0506 FEBRUARY 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Reasoning about connector reconfiguration I: Equivalence of constructions

ABSTRACT
Software systems evolve over time. To facilitate this evolution, the coordination language Reo offers operations to dynamically reconfigure the topology of component connectors. This paper is the first in a series which presents techniques for reasoning about reconfiguration in Reo. This paper presents algebraic laws relating connector construction and reconfiguration. The following paper in the series presents a logic for reasoning about connector behaviour in the presence of dynamic reconfiguration, along with its model checking algorithm.

2000 Mathematics Subject Classification: none
1998 ACM Computing Classification System: D.1.3, D.2.4, F.3.1, F.3.3
Keywords and Phrases: dynamic reconfiguration; coordination languages; Reo
Note: This work was carried out under project SEN3-C-Quattro.

Reasoning about Connector Reconfiguration I: Equivalence of Constructions

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

February 28, 2005

Abstract

Software systems evolve over time. To facilitate this evolution, the coordination language Reo offers operations to dynamically reconfigure the topology of component connectors. This paper is the first in a series which presents techniques for reasoning about reconfiguration in Reo. This paper presents algebraic laws relating connector construction and reconfiguration. The following paper in the series presents a logic for reasoning about connector behaviour in the presence of dynamic reconfiguration, along with its model checking algorithm.

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 the technology to address this issue, providing the glue for plugging software components together. The channel-based coordination language Reo [2] 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, with channels corresponding to wires through which data flows; the channels are connected at "nodes" which regulate data flow; and components are connected at the boundary. Reconfiguration in this setting corresponds to changing the connector between components, so connectors will be our focus.

Just as electronic circuits can, if you are brave enough, be dynamically rewired (reconfigured), so can Reo connectors. The semantics of reconfiguration is clear: before reconfiguration behave like the initial circuit, after reconfiguration behave like the subsequent circuit. Without the proper precautions, rewiring a live electrical circuit is dangerous. An analogous danger exists with running software. For example, data sent to one component may not be received by its intended recipient, or more generally, that the interleaving of reconfiguration steps and data flow may violated a component's expected protocol. Reo's architecture aims to avoid some of this danger, because it can guarantee that certain operations are performed atomically, but it cannot cover all possibilities, such as avoiding protocol faults.

Reasoning about system evolution requires formal models and logics, which this series of papers addresses in a coordination context. To wit, the present paper presents laws relating Reo's construction/reconfiguration operations. The second paper in this series follows on from this work by presenting a modal logic, its semantics and model checking algorithm for reasoning about the behaviour of Reo connectors in the presence of reconfiguration.

Organisation The paper is organised as follows. In the remainder of this introduction, we describe the semantics of Reo connectors and motivate the need for reconfiguration with a simple scenario. Section 2 gives a formalisation of the structure of a Reo connector which is used as the basis for formalising the reconfiguration operations. Section 3 formalises the reconfiguration

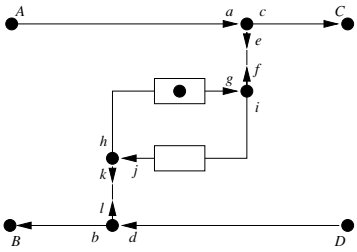


Figure 1: Connector joining Bidder and Auction Components. The Bidder is connected to nodes A (bid) and B (response). The Auction component is connected to nodes C and D.

operations, called *constructions*, and characterises well-formed constructions by defining a notion of *arity*. Section 4 formalises a collection of sound axioms for determining the equivalence of two constructions. Section 5 gives an account of some related work and Section 6 concludes and indicates some directions for future work.

1.1 Reo Overview

Reo is a channel-based coordination language which defines circuit-like *connectors* for coordinating software components [2]. (For a comprehensive introduction, the paper by Arbab [2] is best consulted.) Various kinds of channel are possible, offering different synchronisation, buffering, lossy and even directionality policies. Each channel imposes synchronisation or exclusion constraints on its channel ends, as well as permitting data flow at the ends. Channels are connected at *nodes* which help route data through a connector. A node may have any number of channel ends which can push data into the node and any which can accept data from a node. Data flows at a node if (1) one of the data suppliers (a component writing or an output end of a channel) can succeed *at the exclusion of all others*, and (2) *all* acceptors (a component taking or an input end of a channel) can *synchronously* accept that data. The synchronisation and exclusion constraints imposed by nodes and channels are propogated through the entire connector. This leads to a powerful language for expressing component connectors [2, 1]. One such example is given in Figure 1. This connector involves three channel types. An ordinary arrow, for example A-a, represents a *synchronous channel*, which sends data from one end to another, synchronously. An arrow with a box, such as i-j, corresponds to a *FIFO buffer of size one*. It allows a write to an empty buffer to succeed, and take from a full buffer to succeed. The FIFO1 buffer with a bullet, for example h-g, contains data. An arrow with two arrow heads pointing inwards, such as e-f, is a *synchronous drain*. Two writes to the ends must occur synchronously, with the data being 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 two loops of FIFO1 buffers in the middle of the connector sequence these two events.

Reo also provides operations for constructing and reconfiguring connectors. These consist of an operation for joining two nodes together, one for splitting a node into two, one for creating new channels, one for hiding a internal node of a connector, preventing it from being reconfigured, and one for forgetting boundary nodes, indicating that they are no longer in use, so that they can be garbage collected, plus others not addressed in this paper. Before describing these operations, in Section 3, we present a simple reconfiguration scenario to motivate the reasoning tools presented in this paper.

1.2 Reconfiguration Scenario

A simple reconfiguration scenario is the following. We start with the connector given in Figure 1. Due to new legislation, the requirement is imposed on all auctions that all bids and their corresponding responses must be logged. To implement this requirement, the additional channels

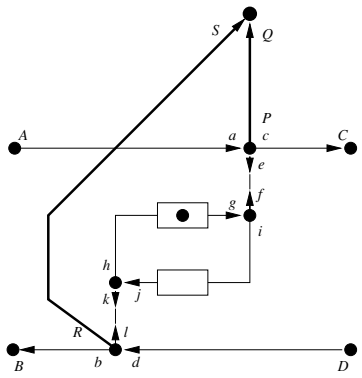


Figure 2: Logging Component added (node SQ). Each bid and response is copied to that node.

highlighted in reconfigured circuit in Figure 2 need to be added, with the Logging component attached at the node SQ.

After a subsequent election, the new government removes the legislation, so the logging part of the connector can be removed. We wish to reason that the series of operations performed on the second connector to remove the logging mechanism does indeed reproduce the original connector. We revisit this scenario in Section 4.3.

We would further like to argue that the behaviour of the resulting circuit is the same as the original, but this will be the topic of the sequel to this paper.

2 Reo Connectors

In this section we present a formal description of the "graph" corresponding to a Reo connector in order to precisely give the semantics of connector constructions and thus describe the effect of reconfiguration operations. Part of the model captures that Reo connectors have both a part which can be observed and reconfigured, and a part which is hidden or forgotten.

Firstly, channels are defined to have both a type, the kind of channel it is, and two ends. 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 receives data (*input end*) or produces data (*output end*).

Definition 2.1 (Channel) *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, a synchronous channel $Ch^{Sync}_{\mathsf{f},\mathsf{g}}$ requires that $io(\mathsf{f}) = i$ and $io(\mathsf{g}) = o$. We will assume throughout that ends are used consistently with their direction.

Connectors are formed by grouping together channel ends into nodes. Thus we represent nodes as sets of channel ends.

Definition 2.2 (Node) *A* node *is a set of channel ends. Let a, b, c, d, e range over nodes. Let ab denote the joining of nodes a and b, defined as $a \cup b$.*

There are a number of different kinds of nodes in a connector. Firstly, there are *boundary nodes*, through which components interact with a connector. Boundary nodes which consist entirely of input ends or output ends are called, respectively, *input nodes* and *output nodes.*[1] An *internal* or *mixed* node of a connector, formally indicated by $mixed(a) \mathrel{\widehat{=}} (a_i \neq a \wedge a_o \neq a)$, can act as a

[1] This terminology differs from the work of Arbab [2] who uses the phrases *source* for *input* and *sink* for *output*. The key for determining which directionality to use is to look from outside the channel/connector. If it receives data it is an input end or node. If it gives data, then it is an output end or node.

self-contained pumping station, making it possible for data to flow between the mixed nodes of a connector, without any external impetus.

The set of channels and the set of nodes over which it is defined, including a statement of which are hidden or forgotten, completely determine a connector, and this will be at the core of our formalisation.

Definition 2.3 (Node Set) *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.*

Connectors are thus built up using channel ends $\mathsf{b} \in \mathcal{E}$, which form nodes $b \in \mathcal{P}(\mathcal{E})$. The collection of nodes forms the node set of a connector, $B \in \mathcal{P}(\mathcal{P}(\mathcal{E}))$. We find it convenient to have access to the sets of ends with a node or a node set.

Definition 2.4 (Underlying Ends) *If a is a node, then $ends(a) = a$. If B is a node set, then $ends(B) = \bigcup_{b \in B} ends(b)$. If $Ch^T_{\mathsf{a},\mathsf{b}}$ is a channel, then $ends(Ch^T_{\mathsf{a},\mathsf{b}}) = \{\mathsf{a}, \mathsf{b}\}$. If Ch is a set of channels, then $ends(Ch) = \bigcup_{x \in Ch} ends(x)$.*

To express constraints on the well-formedness of node sets, and ultimately on constructions, we introduce the following notions of disjointness.

Definition 2.5 (Disjointness) *Let $a\#b$ and $A\#B$ denote that the sets of ends in two nodes a and b or two node sets A and B, respectively, are disjoint. That is, $a\#b \mathrel{\widehat{=}} a \cap b = \emptyset$ and $A\#B \mathrel{\widehat{=}} ends(A) \cap ends(B) = \emptyset$.*

An obvious property of disjoint node sets is that they contain disjoint node sets.

Lemma 2.6 *For node sets A and B, if $A\#B$, then $A \cap B = \emptyset$.*

Finally, we introduce some notation to reduce the need for writing constraints on node sets. Let B_A denote the node set $B \cup A$, with the side condition that $B\#A$. Similarly, B_a denotes $B \cup \{a\}$ such that $B\#\{a\}$.

We now formalise a structural representation of Reo connectors as a collection of its constituent channels, how its ends are grouped to form visible and hidden nodes.

Definition 2.7 (Reo Connector) *A Reo connector $\mathcal{C} = (Ch, B, H)$ consists of a set of channels Ch and a node set B of the* visible nodes*, those which can be reconfigured, and the* hidden node set *H, where $H\#B$. 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 $ends(Ch)$.*

Denote the collection of all connectors by $\mathcal{R}eo$. Let $\mathcal{C}$ and $\mathcal{D}$ range over connectors, and let $=_{\mathcal{R}eo}$ denote equality on connectors, in the sense that if $\mathcal{C} = (Ch_1, B_1, H_1)$ and $\mathcal{D} = (Ch_2, B_2, H_2)$, then $\mathcal{C} =_{\mathcal{R}eo} \mathcal{D}$ if and only if $Ch_1 = Ch_2$, $B_1 = B_2$ and $H_1 = H_2$.

Example 2.8 *The connector in Figure 1(a) is represented by*

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

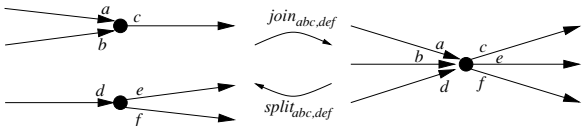


Figure 3: Joining, $\texttt{join}_{abc,def}$, groups nodes *abc* and *def* together to form node *abdcef*. Splitting, $\texttt{split}_{abc,def}$, performs the inverse operation. All possible ways of splitting and joining are permitted. Both operations tend to drastically alter data flow.

3 Constructions on Reo Connectors

Reo is equipped with a language for constructing and reconfiguring connectors [2]. Its essense is captured in the following language of *constructions*, whose syntax is:

$$F, G \quad ::= \quad \texttt{id} \mid GF \mid Ch^{T}_{\mathsf{a},\mathsf{b}} \mid \texttt{join}_{a,b} \mid \texttt{split}_{a,b} \mid \texttt{hide}_{a} \mid \texttt{forget}_{a}$$

Constructions ($\mathcal{C}on$) are operations taking a Reo connector to another Reo connector. That is, a construction $F \in \mathcal{C}on$ is a partial function from $\mathcal{R}eo$ to $\mathcal{R}eo$. The identity construction, $\texttt{id}$, does not modify its argument. Sequential composition of F followed by G, is denoted GF, following the mathematical convention. The 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. *For simplicity we make the assumption that each channel construction appearing within another construction uses distinct channel ends.* The construction $\texttt{join}_{a,b}$ takes two nodes a and b of a connector and joins them together to form a new node ab. The construction $\texttt{split}_{a,b}$ takes a node ab and splits it into two nodes a and b. Split and join are depicted in Figure 3. Finally, $\texttt{hide}_{a}$ takes the mixed node a and $\texttt{forget}_{b}$ takes a non-mixed node (*i.e.,* an input or output node) and prevents it from being reconfigured. Hiding corresponds to making the internal node operate of its own accord, independently of operations at the boundary, if possible, whereas forgetting models that an input or output node is no longer in use, and thus does not contribute to the functioning of a connector. Such nodes may become candidates for garbage collection once all nodes that they are connected to via channels becomes either hidden or garbage. The effect of both these constructions, with respect to reconfiguration, is to make their argument nodes no longer available for further reconfiguration. Note, however that they both have consequences for the behaviour of the resulting connector [3], although we explore this issue no further in this paper.

The action of constructions on Reo connectors is given in the following definition.

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 the partial operation defined recursively as:*

$$\begin{aligned}
\texttt{id}(\mathcal{C}) &= \mathcal{C} \\
GF(\mathcal{C}) &= G(F(\mathcal{C})) \\
Ch^{T}_{\mathsf{a},\mathsf{b}}(Ch, B, H) &= (Ch \cup \{Ch^{T}_{\mathsf{a},\mathsf{b}}\}, B \cup \{\mathsf{a}\} \cup \{\mathsf{b}\}, H), \\
&\qquad \textit{where } \{\mathsf{a}, \mathsf{b}\} \cap \mathit{ends}(B \cup H) = \emptyset \\
\texttt{join}_{a,b}(Ch, B_{a,b}, H) &= (Ch, B_{ab}, H) \\
\texttt{split}_{a,b}(Ch, B_{ab}, H) &= (Ch, B_{a,b}, H) \\
\texttt{hide}_{a}(Ch, B_{a}, H) &= (Ch, B, H_{a}), \qquad \textit{if } \mathit{mixed}(a) \\
\texttt{forget}_{a}(Ch, B_{a}, H) &= (Ch, B, H_{a}), \qquad \textit{if } \neg \mathit{mixed}(a)
\end{aligned}$$

If $F(\mathcal{C})$ is undefined, write $F(\mathcal{C}) = \bot$.

Example 3.2 *The connector in Figure 1 can be built using the construction:*

$$\texttt{join}_{\mathsf{ac},\mathsf{e}}\texttt{join}_{\mathsf{a},\mathsf{c}}\texttt{join}_{\mathsf{gf},\mathsf{i}}\texttt{join}_{\mathsf{g},\mathsf{f}}\texttt{join}_{\mathsf{hj},\mathsf{k}}\texttt{join}_{\mathsf{h},\mathsf{j}}\texttt{join}_{\mathsf{bd},\mathsf{l}}\texttt{join}_{\mathsf{b},\mathsf{d}}$$

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

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

Example 3.3 *A construction for performing the reconfiguration resulting in Figure 2 is:*

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

Example 3.4 *A construction which takes the connector in Figure 2 and reproduces the connector in Figure 1, 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 is more complex and convenient to use [2]. For example, nodes can conveniently be referred to by one of their constituent channel ends, so that end `b` can refer to node {`abc`}. One could also perform a hide on a boundary node in order to freeze it against reconfiguration, or perform a forget on a mixed node again to freeze it against reconfiguration, without imposing the behavioural changes which are a part of hiding. We have chosen not to model these possibilities because they will make the semantics (in the sequel) less clean. Finally, operations are present for connecting and disconnecting components to and from nodes, ensuring access to each node by at most one component at a time. We abstract from these details and from components altogether, in order to capture the essence of connector reconfiguration. Future work will include treatment of these aspects.

3.1 Well-formed Constructions

The goal of this section is to characterise well-formed constructions. Certain constructions are ill-formed because, for example, they make conflicting assumptions about the connectors to which they apply. For example, each of $\mathtt{join}_{a,b}\mathtt{join}_{a,b}$, $\mathtt{join}_{a,b}\mathtt{join}_{a,c}$, and $\mathit{Ch}^{T}_{\mathsf{a,b}}\mathit{Ch}^{T}_{\mathsf{a,c}}$ are ill-formed. Even well-formed constructions cannot be applied to all connectors, because the connector may not satisfy the assumptions made by the construction. To capture well-formedness and the constraints imposed by a well-formed construction, we introduce the notion of *arity*, which describes the conditions on the visible node set under which the construction will work, and indicates any changes made to the visible node set. That is, the arity completely captures the requirements that a connector must satisfy for the construction to make sense when applied to that connector. Indeed much more information can be extracted from an arity, as given in Lemma 3.9. A construction which has no arity is ill-formed.

A valid construction F can be given an *arity* $B \to B'$, where B and B' are nodes sets. The domain node set B states precisely which nodes, from the visible node set, the construction operates on. The codomain node set B' describes which new channel ends are added to the connector and what changes are made to those ends and the nodes in the domain node set. The nodes of a connector not mentioned in the arity are untouched. A construction with arity $B \to B'$ can be applied to any connector whose visible node set contains at least B and does not clash with B'. Thus if F has arity $B \to B'$ it can be thought of as being a polymorphic function of type $\forall C.C_B \to C_{B'}$, including the constraint $C\#B \wedge C\#B'$, which transforms a connector with visible node set C_B to a connector with visible node set $C_{B'}$. Here C corresponds to the nodes which are not touched by the construction.

Before proceeding with the rules for computing arity, we introduce the following notion of overlap, which gives the nodes the in node set A that have some end in common with a node in B, or vice versa.

Definition 3.5 (Overlap) *The overlap of two node sets A and B is defined as:*

$$A \Cap B \mathrel{\widehat{=}} \{a \in A \mid a \cap \mathit{ends}(B) \neq \emptyset\} \cup \{b \in B \mid b \cap \mathit{ends}(A) \neq \emptyset\}$$

The arity of a construction can be computed using the following definition. Note that the composition of two constructions in not always defined, and thus the arity is also not defined. The precise conditions under which it is defined are also given in the definition.

Definition 3.6 (Arity) *The arity of constructions are given inductively as follows:*

- $\mathtt{id} : \emptyset \to \emptyset$;
- $\mathtt{join}_{a,b} : \{a, b\} \to \{ab\}$;
- $\mathtt{split}_{a,b} : \{ab\} \to \{a, b\}$;
- $Ch^T_{\mathtt{a},\mathtt{b}} : \emptyset \to \{\mathtt{a}, \mathtt{b}\}$, *where* $\mathtt{a} \neq \mathtt{b}$.
- $\mathtt{hide}_a : \{a\} \to \emptyset$, *whenever* $mixed(a)$.
- $\mathtt{forget}_a : \{a\} \to \emptyset$, *whenever* $\neg mixed(a)$.
- *The composition* GF *of* $F : A \to A'$ *and* $G : B \to B'$ *is defined whenever:*
  1. $A' \Cap B = A' \cap B$;
  2. $A \cap B \subseteq A'$; *and*
  3. $A' \cap B' \subseteq B$.

  The arity is then $GF : A \cup (B \setminus A') \to B' \cup (A' \setminus B)$.

Let $arity(F)$ *denote the partial function giving the arity of construction* F*, when it is defined.*

The arities of constructions can be explained as follows. The `id` construction is applicable to all connectors, makes no assumptions and makes no changes. Construction $\mathtt{join}_{a,b}$ requires that nodes a and b be present in the connector, and produces a connector which no longer contains a and b, but instead contains the new node ab. Construction $\mathtt{split}_{a,b}$ works in the opposite manner, as expected. Construction $Ch^T_{\mathtt{a},\mathtt{b}}$ makes no assumptions about what is present in a connector, because there is the overriding global assumption that the ends `a` and `b` are fresh, and produces a connector with two new nodes, one for each of the ends. Construction $\mathtt{hide}_a$ requires that the connector has a mixed node a which is then removed from the visible node set of the connector. Construction $\mathtt{forget}_a$ is similar, except that it applies to boundary nodes.

Composition is the most complicated, because our arities are quite abbreviated. Requirement (1) for composition ensures that whenever an end occurs in some node of the domain of F and in some node of the codomain of G, then the two nodes are the same, meaning that the overlap between F and G is consistent. More simply put: the overlap is the same as the intersection. Requirement (2) states that if the node set assumed by both constructions overlaps, then this overlap must be a part of the result of the first construction, because, ultimately, both constructions cannot produce the same channel sets. Requirement (3) boils down to saying that if two constructions produce the same thing, then the second construction G must have it as a requirement, because two composed construction cannot both produce the same channels.

Remark 3.7 *A construction* F *with arity* $\emptyset \to B$ *can be seen as corresponding to a connector. There are only two kinds of connectors with visible node set* $\emptyset$*. The obvious one is the empty connector* $(\emptyset, \emptyset, \emptyset)$*. A less obvious one is a connector* $(Ch, \emptyset, H)$*, which contains no visible nodes. This means that all the channel ends in* Ch *are either hidden or forgotten. Thus no behaviour of the connector can be observed, and this connector ought to be considered as equivalent to* $(\emptyset, \emptyset, \emptyset)$*, because what is present can be considered as garbage—see Section 4.2. Ignoring this garbage, we can consider* $F(\emptyset, \emptyset, \emptyset)$ *to be a connector.*

Arities are unique, when defined, and each end within each node set in the arity belongs to at most one node.

Lemma 3.8 *Assume* $F : A \to A'$.

1. *If* $F : B \to B'$, *then* $A = B$ *and* $A' = B'$.

2. *For all distinct $a, b \in A$, $a\#b$. Similarly for A'.*

Proof: (1) The method for determining the arity is a partial function, thus deterministic. (2) Simple induction on the structure of F, relying on side conditions (1), (2), and (3) of composition. □

Observe that the arity of a construction describes accurately what the structural changes to a connector will be. The only information it abstracts away is the types of new channels.

The following lemma amounts to stating that the arity completely determines whether a construction can be applied to a connector. Furthermore, in the case that the application is defined, the visible node set of the result of applying the construction can be determined from the arities of the construction and its argument. Something can also be said about the nodes which are created and which are hidden.

Lemma 3.9 (Arity Characterisation) *Given construction $F : A \to B$ and a Reo connector $\mathcal{C}$ with visible node set C:*

1. *$F(\mathcal{C})$ is defined, according to Definition 3.1, if and only if there is a visible node set D such $D\#A$, $D\#B$, and $C = A \cup D$. Furthermore, if $F(\mathcal{C})$ is defined, its visible node set is $B \cup D$.*
2. *$ends(B) \setminus ends(A)$ are the new ends created (but not hidden), and $ends(A) \setminus ends(B)$ are the existing ends which become hidden.*

Proof: The proof is rather long and technically uninteresting. It can be found in Appendix A. □

4 Axiomatizing Constructions

Constructions change connectors. As constructions are simple operations on graphs, it is apparent that different constructions may result in the same changes to a connector. In this section we capture this notion, presenting a sound axiomatization of constructions, which determine whether two constructions in all cases produce the same changes to connectors, captured in Definition 4.4. Note that some construction may be equivalent to another over the in domain which it is defined, though it may be defined for fewer connectors. An example is that `id` and $\texttt{split}_{a,b}\texttt{join}_{a,b}$ have the same effect, though the latter requires that the connector have nodes a and b, whereas `id` is defined on all connectors. To account for this partiality, we introduce a partial order between constructions, which is also axiomatized.

We use $F \equiv_{\mathcal{C}on} G$ and $F \sqsubseteq_{\mathcal{C}on} G$ to say that F is provably equivalent to G via the axioms and that F is equivalent to G over the domain in which F is defined, respectively.

The axioms rely on the following notion of independence which describes when two constructions are working on different parts of a connector.

Definition 4.1 (Independence) *$F : A \to A'$ and $G : B \to B'$ are said to be* independent *if and only if $A\#B$ and $A'\#B'$.*

Lemma 4.2 *If F and G are* independent*, then both FG and GF are defined and have the same arity.*

Proof: Assume that $F : A \to A'$ and $G : B \to B'$ are independent. We need to prove firstly that GF and FG are defined. From $A\#B$, that $A\#(ends(B') \setminus ends(B))$, and Lemma 3.9(b), we get $A\#B'$ from which condition (1) of composition follows. Similarly, we can obtain that $A'\#B$. Conditions (2) and (3) of composition (Definition 3.6) are clear since, for example, $A\#B$ implies that $A \cap B = \emptyset$. The arity is then $A \cup B \to A' \cup B'$, because both $B \cap A'$ and $A \cap B'$ are empty. □

The following axioms hold wrt the notion of (partial) equivalence defined above:

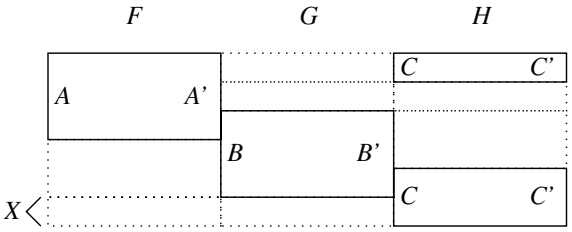
$$
\begin{aligned}
F\mathtt{id} &\equiv_{\mathcal{C}on} F && && (1)\\
F(GH) &\equiv_{\mathcal{C}on} (FG)H && && (2)\\
FG &\equiv_{\mathcal{C}on} GF, && \text{if } F \text{ and } G \text{ are independent} && (3)\\
\mathtt{join}_{a,b} &\equiv_{\mathcal{C}on} \mathtt{join}_{b,a} && && (4)\\
\mathtt{join}_{a,bc}\mathtt{join}_{b,c} &\equiv_{\mathcal{C}on} \mathtt{join}_{ab,c}\mathtt{join}_{a,b} && && (5)\\
\mathtt{split}_{a,b} &\equiv_{\mathcal{C}on} \mathtt{split}_{b,a} && && (6)\\
\mathtt{split}_{b,c}\mathtt{split}_{a,bc} &\equiv_{\mathcal{C}on} \mathtt{split}_{a,b}\mathtt{split}_{ab,c} && && (7)\\
\mathtt{split}_{a,b}\mathtt{join}_{a,b} &\sqsubseteq_{\mathcal{C}on} \mathtt{id} && && (8)\\
\mathtt{join}_{a,b}\mathtt{split}_{a,b} &\sqsubseteq_{\mathcal{C}on} \mathtt{id} && && (9)\\
F &\equiv_{\mathcal{C}on} G, && \text{if } F \sqsubseteq_{\mathcal{C}on} G \wedge \mathit{arity}(F) = \mathit{arity}(G) && (10)
\end{aligned}
$$

The first three axioms express identity (1), associativity (2), and commutativity of independent constructions (3). Axioms (4) and (5) state that the order of elements within a join is irrelevant and that multiple joins forming a given node can be done in any order. Axioms (6) and (7) are analogous for split. Axioms (8) and (9) state that matching split-join pairs can be eliminated. A split-join pair is defined on fewer connectors, because it requires the presence of certain nodes. Finally, Axiom (10) states that the arity of a construction contains enough information to talk about the domain over which it is defined, and thus the arity can be used to infer totality. For example $\mathtt{join}_{a,b}\mathtt{split}_{a,b}\mathtt{join}_{a,b} \equiv_{\mathcal{C}on} \mathtt{join}_{a,b}$. Without Axiom (10), we would have only $\mathtt{join}_{a,b}\mathtt{split}_{a,b}\mathtt{join}_{a,b} \sqsubseteq_{\mathcal{C}on} \mathtt{join}_{a,b}$.

The following lemma states two constructions which are equivalent by our axioms have the same arity. In the case that the two constructions are only partially related, a condition describing their arities is given. We implicitly assume that constructions such as $Ch^T_{\mathtt{a},\mathtt{b}}\mathtt{forget}_{\mathtt{a}}\mathtt{forget}_{\mathtt{b}} : \{\mathtt{a},\mathtt{b}\} \to \{\mathtt{a},\mathtt{b}\}$ are invalid and thus do not cause problems, because such a construction would violate our assumption that channel ends are unique—the two forget statements assume the existence of the ends $\mathtt{a}$ and $\mathtt{b}$, whereas $Ch^T_{\mathtt{a},\mathtt{b}}$ subsequently creates them.

Lemma 4.3 *Given $F : A \to A'$ and $G : B \to B'$, If $F \equiv_{\mathcal{C}on} G$, then $A = B$ and $A' = B'$. Also, if $F \sqsubseteq_{\mathcal{C}on} G$, then $B \subseteq A$ and $B' \subseteq A'$.*

Proof: Most cases are trivial, so are omitted. The case for $FG \equiv_{\mathcal{C}on} GF$ follows from Lemma 4.2. The only remining interesting case is for $F(GH) \equiv_{\mathcal{C}on} (FG)H$. We wish to show $\mathit{arity}(F(GH)) = \mathit{arity}((FG)H)$ according to Definition 3.6. Given the following diagram, it seems that reading from the left gives that the domain visible node set is $A \cup (B \setminus A') \cup ((C \setminus B') \setminus A')$, and from the right gives that the codomain visible node set is $C' \cup (B' \setminus C) \cup ((A' \setminus B) \setminus C)$. We will prove that this is indeed correct.



Assume that $F : A \to A'$, $G : B \to B'$ and $H : C \to C'$. Calculating, we get $GF : A \cup (B \setminus A') \to B' \cup (A' \setminus B)$, and thus

$$H(GF) : A \cup (B \setminus A') \cup (C \setminus (B' \cup (A' \setminus B))) \to C' \cup (B' \cup (A' \setminus B) \setminus C).$$

In the other direction, $HG : B \cup (C \setminus B') \to C' \cup (B' \setminus C)$, and thus

$$(HG)F : A \cup ((B \cup (C \setminus B')) \setminus A') \to C' \cup (B' \setminus C) \cup ((A' \setminus B) \setminus C).$$

We need to show that the domains and codomains are indeed equal. Let's first attack the domain. X corresponds to $C \setminus (B' \cup (A' \setminus B))$ in the diagram. $X = C \setminus (B' \cup (A' \setminus B)) = (C \setminus B') \setminus (A' \setminus B) = (C \cap \overline{B'}) \cap \overline{(A' \cap \overline{B})} = C \cap \overline{B'} \cap (\overline{A'} \cup B) = (C \cap \overline{B'} \cap \overline{A'}) \cup (C \cap \overline{B'} \cap B)$. From requirement (2) of composition, $C \cap \overline{B'} \cap B = \emptyset$, thus $X = C \cap \overline{B'} \cap \overline{A'} = (C \setminus B') \setminus A'$. The domain of $H(GF)$ is therefore $A \cup (B \setminus A') \cup ((C \setminus B') \setminus A')$. Now, taking part of the domain of $(HG)F$ we obtain $(B \cup (C \setminus B')) \setminus A' = (B \setminus A') \cup ((C \setminus B') \setminus A')$, thus the domain of $(HG)F$ is $A \cup (B \setminus A') \cup ((C \setminus B') \setminus A')$, which is the same as the domain of $H(GF)$, as required.

The calculation for the codomain is identical, using requirement (3) of composition. □

4.1 Soundness of the Axiomatization

Definition 4.4 (Construction Equivalence) *Two constructions F and G are equivalent, denoted $F \equiv_{\mathcal{S}em} G$, if and only if for all connectors $\mathcal{C}$, $F(\mathcal{C}) =_{\mathcal{R}eo} G(\mathcal{C})$. As some constructions are defined for more connectors than others, a partial order exists between constructions, $F \sqsubseteq_{\mathcal{S}em} G$, which states that for all connectors $\mathcal{C}$ for which F is defined, $F(\mathcal{C}) =_{\mathcal{R}eo} G(\mathcal{C})$.*

The axioms are sound in the sense that two constructions demonstrably equivalent by the axioms correspond to equal operations on connectors. Similarly, inequality matches partiality.

Theorem 4.5 (Soundness of Axioms) *For all constructions F and G:*

1. *If $F \equiv_{\mathcal{C}on} G$, then $F \equiv_{\mathcal{S}em} G$.*
2. *If $F \sqsubseteq_{\mathcal{C}on} G$, then $F \sqsubseteq_{\mathcal{S}em} G$.*

Proof: For axioms (1) and (2) the desired result holds because function composition is associative and has the identity function as unit. Axiom (3) holds because different parts of the connector are being operated upon by F and G, so their operations do not interfere. Axioms (4)–(9) straightforwardly follow by reasoning about the basic set theoretic operations they correspond to. Axiom (10) is sound because, by induction, $F \sqsubseteq_{\mathcal{C}on} G$ implies that $F \sqsubseteq_{\mathcal{S}em} G$, meaning that they are equal for the domain over which they are defined. Lemma 3.9 states that because F and G have the same arity, they are defined over the same domain. Thus $F \equiv_{\mathcal{S}em} G$. □

We conjecture that these axioms, when extended with the appropriate derivation rules, will be complete.

4.2 Garbage

The last aspect of our formalisation that we cover will be garbage. A connector (or a disjoint part thereof) can be considered as garbage whenever all of its mixed nodes are hidden and all of its boundary nodes are forgotten. The behaviour of such a connector cannot be observed, if it even has behaviour: it has no boundary nodes, nor can the connector be modified in such a way that any of its channels can be reused, (since all of its nodes are hidden or forgotten) and thus no longer accessible to reconfiguration.

Characterising garbage is quite simple. We do so as an equivalence on Reo connectors which corresponds to garbage collection:

$$(Ch \uplus Ch', B, H \uplus H') \quad \equiv_{\mathcal{GC}} \quad (Ch, B, H), \qquad \text{if } ends(Ch') = ends(H')$$

This states that part of the connector constructed using channels Ch' can be considered garbage whenever all of the ends of those channels, and no others, comprise the nodes of some subset of the hidden and forgotten nodes.

From this we can derive a rule for relating two constructions, where one contains garbage, and the other does not contain that garbage. This is really simple, actually:

$$F \quad \equiv_{\mathcal{C}on} \quad \texttt{id}, \qquad \text{if } arity(F) = \emptyset \to \emptyset \tag{11}$$

It is not difficult to see that under the extended notion of equivalence between connectors using $\equiv_{\mathcal{GC}} \cup =_{\mathcal{R}eo}$, the above axiom is sound.

In the sequel to this paper we will show that the presence of garbage has no effect on the behaviour of a connector and that it may be safely eliminated at any time.

4.3 Scenario 1, Revisited

We are now in a position to show that the construction given in Example 3.3 to add logging to the bidder-auction connector could be eliminated using the construction given in Example 3.4 to reproduce the original connector. The axioms we have given enable us to show that the composition of the two constructions is equivalent (modulo garbage collection) to the identity construction:

$$
\begin{array}{ll}
 & \mathtt{forget}_{\mathsf{P}}\mathtt{forget}_{\mathsf{R}}\mathtt{forget}_{\mathsf{SQ}}\underline{\mathtt{split}_{\mathsf{P,ace}}\mathtt{split}_{\mathsf{R,bd|}}\mathtt{join}_{\mathsf{P,ace}}\mathtt{join}_{\mathsf{R,bd|}}}\mathtt{join}_{\mathsf{S,Q}}\, Ch^{Sync}_{\mathsf{R,S}}\, Ch^{Sync}_{\mathsf{P,Q}} \\
 & \{ \text{ Commutativity (3) and axiom (8) twice. } \} \\
\equiv_{\mathcal{C}on} & \mathtt{forget}_{\mathsf{P}}\mathtt{forget}_{\mathsf{R}}\mathtt{forget}_{\mathsf{SQ}}\mathtt{join}_{\mathsf{S,Q}}\, Ch^{Sync}_{\mathsf{R,S}}\, Ch^{Sync}_{\mathsf{P,Q}} \\
 & \{ \text{ Garbage Collection (11) } \} \\
\equiv_{\mathcal{C}on} & \mathtt{id}
\end{array}
$$

Note that the last step relies on the implicit assumption that any new channels created will be distinct. This is roughly equivalent to α-conversion: the names of the hidden elements, and in particular, of the garbage, shouldn't matter.

5 Related Work

Technology for reconfigurable systems has been realised in many different guises: mobile agents, dynamic rebinding of libraries [4], component-hot swapping, and via a coordination layer, whether it be a tuple space [9], a tool bus [5], or some form of component connector [2, 11, 10]. 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 [8], Klaim [6], the lambda calculus of dynamic rebinding [7], and so on. The present work is the first we are aware of for reasoning about the reconfiguration of Reo software connectors.

6 Conclusion and Future Work

This paper presented a formalism for reasoning about Reo connectors in the presence of reconfiguration, by presenting a set of laws which determine whether two constructions are equivalent, in that they have the same effect on all connectors. The results presented here are thus useful for reasoning about the evolution of systems connected by Reo connectors.

A shortcoming of the present work, however, is that it does not consider the behaviour of the connectors. When interleaved with input-output behaviour, equivalences described in this report may no longer hold, because, for example, the contents of buffers may change. Indeed, dynamically reconfiguring an active Reo connector is fraught with danger, because without precautionary measures being taken, almost anything can go wrong, since almost anything can change. We address this issue in the sequel to this paper.

Acknowledgement Many thanks to Helle Hvid Hansen whose comments helped significantly improve this paper. She even proved some of the cases of Lemma 4.3. Farhad Arbab clarified some issues regarding Reo and proofread the paper.

References

[1] Farhad Arbab. Abstract behavior types: A foundation model for components and their composition. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 33–70. Springer, 2003.

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

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] Stépane 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.

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

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

A Proof of Lemma 3.9

Proof: We first prove part 1.

Case $F = \mathtt{id} : \emptyset \to \emptyset$. $F(\mathcal{C})$ is always defined and $D = C$.

Case $F = Ch^T_{\mathtt{a},\mathtt{b}} : \emptyset \to \{\mathtt{a}, \mathtt{b}\}$. $F(\mathcal{C})$ is defined if and only if $C \# \{\mathtt{a}, \mathtt{b}\}$, by Definition 3.1. Setting $D = C$, we obtain correctly $C \cup \{\mathtt{a}, \mathtt{b}\}$ as the free node set for $F(\mathcal{C})$.

Case $F = \mathtt{join}_{a,b} : \{a, b\} \to \{ab\}$. $F(\mathcal{C})$ is defined if and only if $\{a, b\} \subseteq C$ and $C \# \{ab\}$. Setting $D = C \setminus \{a, b\}$, we obtain correctly $C \cup \{ab\}$ as the free node set for $F(\mathcal{C})$.

Case $F = \mathtt{split}_{a,b}$. Argument is analogous to the $\mathtt{join}_{a,b}$ case.

Case $F = \mathtt{hide}_a : \{a\} \to \emptyset$. $F(\mathcal{C})$ is defined if and only if $a \in C$. Let $D = C \setminus a$, and we have the desired result.

Case $F = \mathtt{forget}_a$. Argument is analogous to the $\mathtt{hide}_a$ case.

Case $F = GH : A \to B$.

($\Longrightarrow$) Assume that $F(\mathcal{C})$ is defined. Then, $H(\mathcal{C})$ and $G(H(\mathcal{C}))$ are defined. Assume $H : A' \to B'$, then by the induction hypothesis, there is a D' such that $D'\#A'$ and $D'\#B'$, with $D' \cup A' = C$ and $D' \cup B' = \mathit{freeNodes}(H(\mathcal{C}))$. Assume $G : A'' \to B''$, then by the induction hypothesis, there is a D'' such that $D''\#A''$ and $D''\#B''$ and $D'' \cup A'' = \mathit{freeNodes}(H(\mathcal{C}))$ and $D'' \cup B'' = \mathit{freeNodes}(G(H(\mathcal{C}))) = \mathit{freeNodes}(F(\mathcal{C}))$. Note further that, by the arity rules for composition, we must have $A = A' \cup (A'' \setminus B')$ and $B = B'' \cup (B' \setminus A'')$, as well as the side conditions: $A' \cap A'' \subseteq B'$ and $B' \cap B'' \subseteq A''$.

We now want to show that there exists a D such that $D\#A$, $D\#B$, and $D \cup A = D' \cup A' = C$, and $D \cup B = D'' \cup B''$. There are two canditates D, namely $D' \setminus A''$ and $D'' \setminus B'$. These are equivalent because $D' \uplus B' = D'' \uplus A''$. Further, observe that $A'' \setminus B' \subseteq D'$, because from $D' \uplus B' = D'' \uplus A''$ we can deduce that $D' = (D'' \uplus A'') \setminus B' = (D'' \setminus B') \uplus (A'' \setminus B') \supseteq A'' \setminus B'$.

Now $D\#A$, as this is equivalent by definition to $(D' \cap \overline{A''})\#(A' \cup (A'' \setminus B'))$. This holds if and only if $(D' \cap \overline{A''})\#A''$ and $(D' \cap \overline{A''})\#(A'' \setminus B')$ hold. The first of these holds because $D'\#A'$. The second holds because $\overline{A''}\#A$.

Secondly, $D \cup A = D' \cup A'$. $D \cup A$ is equivalent by definition to $(D' \setminus A'') \cup A' \cup (A'' \setminus B')$. Note that $D' \setminus A'' \equiv D' \setminus (A'' \setminus B')$, since $D'\#B'$. So, the above is equivalent to $D' \cup A' \cup (A'' \setminus B')$. Now as $A'' \setminus B' \subseteq D'$, we obtain $D' \cup A'$.

The cases for $D\#B$ and $D \cup B = D'' \cup B''$ are similar. This completes one direction.

($\Longleftarrow$) Assume that $GH : A \to B$ and that there exists a D such that $D\#A$, $D\#B$ and $C = A \cup D$. From the arity calculations, we can determine that there are A', B', A'' and B'' such that $G : A'' \to B''$ and $H : A' \to B'$, with $A = A' \cup (A'' \setminus B')$ and $B = B'' \cup (B' \setminus A'')$.

Let $D' = D \cup (A'' \setminus B')$. We want to show (1) $D' \cup A' = D \cup A$, (2) $D'\#A'$, and (3) $D'\#B'$.

1. $D' \cup A' = D \cup (A'' \setminus B') \cup A' = D \cup A$.

2. $D\#A$ and $A = A' \cup (A'' \setminus B')$ implies that $D\#A'$. Further, $A' \cap A'' \subseteq B'$ implies that $(A'' \setminus B')\#A'$. Thus $D \cup (A'' \setminus B')\#A'$, establishing $D'\#A'$.

3. Observe that $B' = (B' \setminus A'') \cup (B' \cap A'')$. $D\#B \Longleftrightarrow D\#(B'' \cup (B' \setminus A''))$, giving $D\#(B' \setminus A'')$ and $D\#B''$. It is clear that $\mathit{ends}(A'') \subseteq \mathit{ends}(B'') \cup \mathcal{H}$, where $\mathcal{H}$ contains the ends of A'' hidden by G. By the assumption that each channel creation creates new end names, we must have that $\mathcal{H}\#D$, as otherwise this would imply that there were two creators of such end names. Thus we obtain that $D\#A''$. Hence $D\#(B' \cap A'')$, which with $D\#(B' \setminus A'')$ gives $D\#B'$. Now $D' = D \cup (A'' \setminus B')$. Clearly, $(A'' \setminus B')\#B'$, thus $D'\#B'$.

Now we are in a position to invoke the induction hypothesis and obtain that $H(\mathcal{C})$ is well defined with free node set $D' \cup B'$. Let $D'' = D \cup (B' \setminus A'')$. Arguments analogous to those above enable us to establish $D \cup B = D'' \cup B$, $D''\#A''$ and $D''\#B''$. We now need to show that $D'' \cup A'' = D' \cup B'$. This is straightforward: $D'' \cup A'' = D \cup (B' \setminus A'') \cup A'' = D \cup (B' \setminus A'') \cup (A'' \cap B') \cup (A'' \setminus B') = D \cup B' \cup (A'' \setminus B') = D' \cup B'$. Again, we can invoke the induction hypothesis to obtain that $G(H(\mathcal{C}))$ is defined with free node set $D'' \cup B''$.

Now all we need show is that $D'' \cup B'' = D \cup B$, and we are done. This is simple: $D'' \cup B'' = D \cup (B' \setminus A'') \cup B'' = D \cup B$.

This now completes part 1 of the theorem.

For Part 2, the only challenging case is again composition.

Case $F = GH : A \to B$. From the arity calculations, we can determine that there are A', B', A'' and B'' such that $G : A'' \to B''$ and $H : A' \to B'$, with $A = A' \cup (A'' \setminus B')$ and $B = B'' \cup (B' \setminus A'')$.

By induction, $\mathit{ends}(B') \setminus \mathit{ends}(A')$ are the new ends created (but not hidden), after applying H. By induction, $\mathit{ends}(B'') \setminus \mathit{ends}(A'')$ are the new ends created (but not hidden), after applying G. Thus the new ends created are those created by either G or H, but not hidden by G, which is $(\mathit{ends}(B') \setminus \mathit{ends}(A')) \setminus (\mathit{ends}(A'') \setminus \mathit{ends}(B'')) \cup (\mathit{ends}(B'') \setminus \mathit{ends}(A'')$. Now abbreviating the ends sets by the corresponding lower case letter, we obtain

$$\begin{aligned} & ((b' \setminus a') \setminus (a'' \setminus b'')) \cup (b'' \setminus a'') \\ = \;& ((b' \cap \overline{a'}) \cap \overline{(a'' \cap \overline{b''})}) \cup (b'' \cap \overline{a''}) \\ = \;& ((b' \cap \overline{a'}) \cap (\overline{a''} \cup b'')) \cup (b'' \cap \overline{a''}) \end{aligned}$$

$$
\begin{array}{ll}
= & (((b' \cap \overline{a'}) \cap (\overline{a''} \cup b'')) \cup b'') \cap (((b' \cap \overline{a'}) \cap (\overline{a''} \cup b'')) \cup \overline{a''}) \\
= & (((b' \cap \overline{a'}) \cup b'') \cap (\overline{a''} \cup b'' \cup b'')) \cap (((b' \cap \overline{a'}) \cup \overline{a''}) \cap (\overline{a''} \cup b'' \cup \overline{a''})) \\
= & (((b' \cap \overline{a'}) \cup b'') \cap (\overline{a''} \cup b'')) \cap (((b' \cap \overline{a'}) \cup \overline{a''}) \cap (\overline{a''} \cup b'')) \\
 & \{ \text{ By (2) and (3) of composition. } \} \\
= & ((b' \cap \overline{a'}) \cup b'') \cap (\overline{a''} \cap (\overline{a''} \cup b''))
\end{array}
$$

Now condition (1) of composition, which states that the overlap (wrt ends) is the same as the intersection (wrt nodes) enables this to be converted to the desired conclusion:

$$ends(B'' \cup (B' \setminus A'')) \setminus ends(A' \cup (A'' \setminus B')) \quad = \quad ends(B) \setminus ends(A)$$

By induction $ends(A') \setminus ends(B')$ are the existing ends which become hidden when applying H. Similarly, $ends(A'') \setminus ends(B'')$ are the existing ends which become hidden when applying G. Thus the ends which become hidden when applying GH are $(ends(A') \setminus ends(B')) \cup (ends(A'') \setminus ends(B''))$, less the ends which are introduced by G, namely $ends(B') \setminus ends(B')$.

A calculuation similar to the one above obtains:

$$
\begin{array}{ll}
 & ((ends(A') \setminus ends(B')) \cup (ends(A'') \setminus ends(B''))) \setminus (ends(B') \setminus ends(B')) \\
= & ends(A' \cup (A'' \setminus B')) \setminus ends(B'' \cup (B' \setminus A'')) \\
= & ends(A) \setminus ends(B)
\end{array}
$$

□