Reo: A Channel-based coordination model for component composition

Article *in* Mathematical Structures in Computer Science · June 2004

DOI: 10.1017/S0960129504004153 · Source: DBLP

1 author:

Farhad Arbab
Centrum Wiskunde & Informatica
327 PUBLICATIONS **6,831** CITATIONS

SEE PROFILE

Under consideration for publication in Math. Struct. in Comp. Science

Reo: A Channel-based Coordination Model for Component Composition

Farhad Arbab

email: farhad@cwi.nl

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Received 8 June 2003

In this paper, we present Reo, a paradigm for composition of software components based on the notion of mobile channels. Reo is a channel-based exogenous coordination model wherein complex coordinators, called *connectors* are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. Reo can be used as a language for coordination of concurrent processes, or as a "glue language" for compositional construction of connectors that orchestrate component instances in a component-based system. The emphasis in Reo is on connectors and their composition only, not on the entities that connect to, communicate, and cooperate through these connectors. Each connector in Reo imposes a specific coordination pattern on the entities (e.g., components) that perform I/O operations through that connector, without the knowledge of those entities. Channel composition in Reo is a very powerful mechanism for construction of connectors. We demonstrate the expressive power of connector composition in Reo through a number of examples. We show that exogenous coordination patterns that can be expressed as (meta-level) regular expressions over I/O operations can be composed in Reo out of a small set of only five primitive channel types.

1. Introduction

Modular design and construction of software involves modules that rather intimately know and rely on each other's interfaces and fit together like pieces in a jigsaw puzzle. In contrast, software components are expected to be more independent of each other and the specific application environments wherein they are deployed. Because modules can be less independent of their application environments, the provisions for the required interfacing among them can be designed into the modules that make up a modular system. However, if the functionality of each such module is to be supported by a component instead, the bulk of this interfacing must be left out of the individual components, because provisions for interfacing of a component depend on the context wherein it is deployed and the other components that it may interact with. The components that comprise a system, thus, typically do not exactly fit together as pieces of a jigsaw puzzle: they leave significant interfacing gaps that must somehow be filled with additional code. Such interfacing code

is often referred to as "glue code" and is typically highly special purpose and specific. Simplified programming languages, sometimes called *scripting languages*, are often used to write such glue code.

The (scripting) programs that constitute the glue code are inherently no different than other software. In complex systems, the bulk of the specialized glue code by itself can grow in its size and rigidity, rendering the system hard to evolve and maintain, in spite of the fact that this inflexible code wraps and connects reusable, maintainable, and replaceable components.

An alternative to writing scripts or specialized glue code is to construct the glue code compositionally, out of primitive connectors. A promising approach in this direction is to use channels as the primitives out of which such connectors are constructed. Reo defines the primitive operations that allow composition of channels into complex connectors.

A channel is a point-to-point medium of communication with its own unique identity and two distinct ends. Channels can be used as the only primitive constructs in communication models for concurrent systems. Like the primitive constructs in other communication models, channels provide the basic temporal and spatial decouplings of the parties in a communication, which are essential for explicit coordination. Channel-based communication models are "complete" in the sense that they can easily model the primitives of other communication models (e.g., message passing, shared spaces, or remote procedure calls). Furthermore, channel-based models have some inherent advantages over other communication models, especially for concurrent systems that are distributed, mobile, and/or whose architectures and communication topologies dynamically change while they run:

— **Efficiency**: Like remote procedure calls and message passing, channel-based models support point-to-point communication. As such, in contrast to shared data space models, the intended target of communication is always unique and internally known to the system. In truly distributed systems, this allows more efficient implementations of point-to-point models.

— **Security**: In shared data space models, the data in every communication (if not its actual information content) is always exposed for everyone to observe and consume. Furthermore, third parties can, accidentally or intentionally, produce data that look like, and thus may get co-opted as, the data of some particular communication. In contrast, point-to-point models shield communication from accidental exposure to or intentional interference by third parties.

— **Architectural Expressiveness**: Figure 1 shows examples of the connections among component instances (represented as boxes) using three different communication models. In this figure, channels and direct connections are shown as straight lines; the shared data space is shown as an amorphous blob; and the software bus is shown as an elongated rectangle. A point-to-point communication model of an application (Figure 1.a) represents its communication pattern and is highly expressive of its architecture: in such a model, it is clear to see which other components or entities can possibly be affected if a given component or entity is modified or replaced. Models such as shared data spaces (Figure 1.b) and software buses (Figure 1.c) are not architecturally expressive because they contain no explicit representation of such relevant

information as which specific components or entities actually communicates with each other.

— **Anonymity**: Anonymous communication means that the parties involved in a communication need not necessarily know each other. In contrast to remote procedure calls or message passing models, channel-based models can support the anonymous communication which is one of the hallmarks of shared data space models.

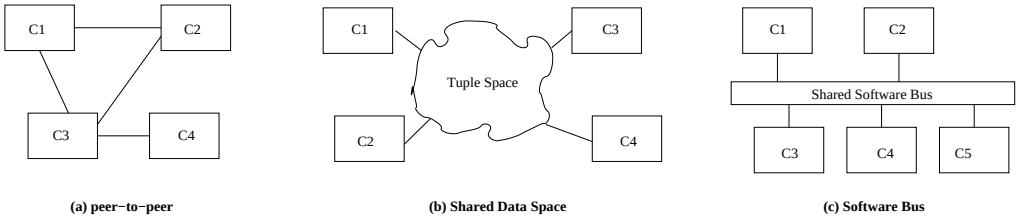

Fig. 1. Architectural expressiveness

The characteristics of channel-based models are attractive from the point of view of coordination. Dataflow models, Kahn networks (Kahn, 1974), and Petri-nets can be viewed as specialized channel-based models that incorporate certain basic constructs for primitive coordination. IWIM (Arbab, 1996; Katis et al., 2000) is an example of a more elaborate coordination model based on channels, and Manifold (Bonsangue et al., 2000) is an incarnation of IWIM as a real coordination programming language that supports dynamic reconfiguration of Kahn network topologies.

A common strand running through these models is a notion that is called "exogenous coordination" in IWIM (Arbab, 1998). This is the concept of "coordination from outside" the entities whose actions are coordinated. Exogenous coordination is already present, albeit in a primitive form, in dataflow models: unbeknownst to a node, its internal activity is coordinated (or, in this primitive instance, merely synchronized) with the rest of the network by the virtue of the input/output operations that it performs. IWIM and Manifold allow much more sophisticated exogenous coordination of active entities in a system.

In this paper we describe Reo, a channel-based model for exogenous coordination introduced in (Arbab and Mavaddat, 2002). The name Reo is pronounced "rhe-oh" and comes from the Greek word $\rho\epsilon\omega$ which means "*[I] flow*" (as water in streams and channels). In plain English text, $\rho\epsilon\omega$ is best transcribed as *Reo*.

Our work on Reo builds upon the IWIM model of coordination and the coordination language Manifold, and extends our earlier work on components. In (Arbab et al., 2000a) a language for dynamic networks of components is introduced, and in (de Boer and Bonsangue, 2000) a compositional semantics for its asynchronous subset is given. A formal model for component-based systems is presented in (Arbab et al., 2000b), together with a formal-logic-based component interface description language that conveys the observable semantics of components, a formal system for deriving the semantics of a composite system out of the semantics of its constituent components, and the conditions under which this derivation system is sound and complete. A concrete incarnation of mobile channels

to support our formal model for component-based systems is presented in (Scholten, 2001). Generalization of data-flow networks for describing dynamically reconfigurable or mobile networks has also been studied in (Broy, 1995) and (Grosu and Stoelen, 1996) for a different notion of observables using the model of stream functions.

Reo is based on a calculus of channels wherein complex connectors are constructed through composition of simpler ones, the simplest connectors being an arbitrary set of channels with well-defined behavior. Reo can be used as the "glue code" in Component Based Software Engineering, where a system is compositionally constructed out of components that interact and cooperate with each other anonymously through Reo connectors.

The rest of this paper is organized as follows. The basic concepts of components, connectors, channels, etc. are introduced in Section 2. What Reo expects from channels is described in Section 3. Most of the channel operations defined in Section 3 are not to be used in the (instances of) components directly; they are low-level operations that are used internally by Reo to define its higher-level operations on connectors. Connectors and channel composition are discussed in Section 4. Patterns and channel types are described in Sections 5 and 6, respectively. Sections 7, 8, and 9 provide an insight into the operational semantics of Reo with hints of its actual implementation. Section 10 contains a number of examples of simple connectors constructed out of channels. In Section 11 the expressiveness of the compositional paradigm of Reo is demonstrated through a number of more complex connectors that can be used to implement any coordination pattern that can be expressed as a regular expression over channel input/output operations. In contrast to the informal operational semantics described in Sections 7, 8, and 9, Section 12 contains an overview of a particularly interesting formal coalgebraic semantics for Reo. Finally, a summary of our conclusions and future work is presented in Section 13.

2. Basic Concepts

Reo is a coordination model and as such has very little to say about the computational entities whose activities it coordinates. These entities can be fragments or modules of sequential code, passive or active objects, threads, processes, agents, or software components. Without loss of generality, we refer to these entities as *component instances* in Reo. From the point of view of Reo, a system consists of a number of component instances executing at one or more locations, communicating through *connectors* that coordinate their activities. This is shown in Figure 2, where component instances are represented as boxes, channels as straight lines, and connectors are delineated by dashed lines. Each **connector** in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of channels. This is why each dashed closed curve representing a connector in Figure 2 contains only a set of channels connected together in a specific topology.

A **component instance**, p, is a non-empty set of active entities (e.g., processes, agents, threads, actors, etc.) whose only means of communication with the entities outside of this set is through input/output operations that they perform on a (dynamic) set of channel ends that are *connected* to p. The communication among the active entities

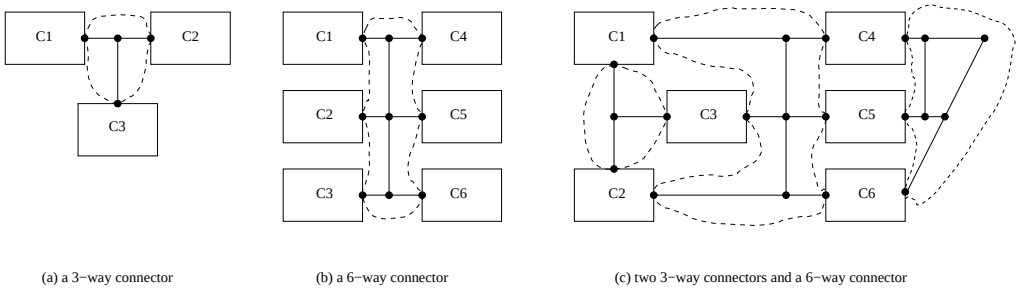

Fig. 2. Components and connectors

inside a component instance, and the mechanisms used for this communication, are of no interest. Likewise, Reo is oblivious to the synchronization, mutual exclusion, and coordination that may have to take place among the active entities inside a component instance for their proper utilization of the channel ends that are connected to that component instance. All these details are internal to a component instance and, thus, irrelevant. What is relevant is only the inter-component-instance communication which takes place exclusively through channels that comprise Reo connectors. Indeed, the constituents inside a component instance may themselves be other component instances that are connected by Reo connectors.

A **component** is a software implementation whose instances can be executed on physical or logical devices. Thus, a component is an abstract type that describes the properties of its instances.

A physical or logical device where an active entity executes is called a **location**. Examples of a location include a Java virtual machine; a multi-threaded Unix process; a machine, e.g., as identified by an IP address; etc. A component instance may itself be distributed, in the sense that its constituents may be executing at different locations (in which case, this too is an internal detail of the component instance to which Reo is oblivious). Nevertheless, there is always a unique location associated with every (distributed) component instance, indicating where that component instance is (nominally) located. There can be zero or more component instances executing at a given location, and component instances may migrate from one location to another while they execute (mobility). As far as Reo is concerned, the significance of a location is that inter-component communication may be cheaper among component instances that reside at the same location.

The only primitive medium of communication between two component instances is a **channel**, which represents an atomic connector in Reo. A channel has its own unique identity. Channels are dynamically created in Reo and they are automatically garbage collected; i.e., they are not explicitly destroyed.

A channel itself has no direction, but each channel in Reo has exactly two directed ends, with their own identities, through which components refer to and manipulate that channel and the data it carries. There are two types of channel ends: sources and sinks. A **source** channel end accepts data into its channel. A **sink** channel end dispenses data out of its channel. A channel end that is known to a component instance can be used by any of the active entities inside that component instance in Reo operations.

Channels are used in Reo exclusively to transfer data using input/output operations performed on their ends (specifically, observe that channels do *not* support "message passing" with the "method-call" semantics). A subset of Reo operations (e.g., the input/output operations) can be performed by (an entity inside) a component instance on a channel end, only if the channel end is connected to that component instance. The identity of a channel-end may be known to zero or more component instances, but each channel end can be **connected** to at most one component instance at any given time. The connection of a channel end to a component instance is a logical notion that is independent of the locations of the channel end and the component instance. The active entities inside a component instance that shares the same location with one of its connected channel ends may be able to more efficiently manipulate that channel end, but co-location (of component instances and their connected channel ends) is not a prerequisite for any such operation.

Both components and channels are assumed to be **mobile** in Reo. A component instance may move from one location to another during its lifetime. When this happens, the channel ends connected to this component instance remain connected, preserving the topology of channel connections. Furthermore, a channel end connected to a component instance may be moved by the active entities inside that component instance to another location, perhaps to enhance the efficiency of subsequent operations on this channel end, still preserving the topology of channel connections. Irrespective of locations, a channel end connected to a component instance may be disconnected from that component instance, and connected to another component instance. This, of course, dynamically changes the topology of channel connections in the system.

3. Primitive Channel Operations

The set of primitive operations on channels and channel ends in Reo is summarized in Table 1. The names of all these operations begin with an underscore because they are to be used internally by Reo only: (the active entities inside) component instances are not allowed to perform these operations directly.

The first column in Table 1 gives the syntax of the operations. Italics denote meta-symbols. Square brackets are meta-symbols that indicate optional parameters. The argument *chantype* designates a channel type, e.g., one of the identifiers that appear in Tables 6 and 7.

— The parameter *loc* identifies a location.
— The parameter *e* stands for a channel-end-value, which is either a source or a sink end of a channel.
— The optional parameter *t* indicates a time-out value greater than or equal to 0. When no time-out is specified for an operation, it defaults to ∞. An operation returns with a result that indicates failure if it does not succeed within its specified time-out period.
— The parameter *conds* is a channel wait condition expression described in Section 3.6.
— The parameter *inp* is a sink of a channel, from which data items can be obtained.
— The parameter *outp* is the source of a channel, into which data items can be written.

Operation	Con.	Description
`_create(`*chantype*`)`	-	Creates a channel of type *chantype* and returns the identifiers of its two channel ends.
`_forget(`*e*`)`	N	changes *e* such that it no longer refers to the channel end it designates.
`_move(`*e, loc*`)`	Y	moves *e* to the location *loc*.
`_connect(`*[t,] e*`)`	N	Connects the specified channel end, *e*, to the component instance that contains the active entity that performs this operation.
`_disconnect(`*e*`)`	N	Disconnects the specified channel end from the component instance that contains the active entity that performs this operation.
`_wait(`*[t,] conds*`)`	N	Suspends the active entity that performs this operation, waiting for the conditions specified in *conds* to become true for the specified channel ends.
`_read(`*[t,] inp[, v[, pat]]*`)`	Y	Suspends the active entity that performs this operation, waiting for a value that can match with *pat*, to become available for reading from the sink channel end *inp* into the variable *v*. The `_read` operation is non-destructive: the value is copied from the channel into the variable, but the original remains intact.
`_take(`*[t,] inp[, v[, pat]]*`)`	Y	This is the destructive variant of `_read`: the channel loses the value that is read.
`_write(`*[t,] outp, v*`)`	Y	Suspends the active entity that performs this operation, until it succeeds to write the value of the variable *v* to the source channel end *outp*.

Table 1. *Primitive channel operations*

— The parameter *v* is a variable from/into which a data item is to be transferred into/from the specified channel end.
— The parameter *pat* is a pattern (see Section 5) that must match with a data item for it to be transferable to *v*.

The second column in Table 1 indicates whether or not a connection between the component instance and the channel end involved in an operation is a prerequisite for the operation. Clearly, this is irrelevant for the `_create` operation. The operations `_forget`, `_connect`, `_disconnect`, and the conditions in `_wait` can specify any channel end irrespective of whether or not it is connected to the component instance involved. The `_move` and the I/O operations `_read`, `_write`, and `_take`, on the other hand, fail if the active entities that perform them reside in component instances that are not connected to the channel ends involved in these operations.

Every channel type in Reo must support the primitive operations in Table 1, with a "reasonable variation" of the semantics for each operation as described below. We allow "reasonable variations" in the precise semantics of these primitives because we wish to allow for such varieties of channels as "read-only" channels, "immutable" channels, and "lossy channels" each of which may require slight deviations in the exact semantics of how some of these operations are supported. For instance, a read-only channel may not support the destructive effect of `_take`, an immutable channel may not allow destruction

or modification of the data items it contains through `_take` and `_write`, and a lossy channel may throw away certain data items at the time of their `_write`, etc.

As far as Reo is concerned, the effect of concurrently performing more than one I/O operation on a channel end is undefined. For instance, if an I/O operation is already pending on a channel end e and another active entity performs another I/O operation on e, the channel to which e belongs can reject the second operation as an error, preempt the first operation and perform the second, serialize the operations in some order, etc., or even even behave unpredictably. Reo simply does not depend on this aspect of the behavior of its primitive channels. See Section 4.1.3 for concurrent I/O operations on nodes.

3.1. *Channel* `_create`

The `_create` operation creates a channel and returns the identifiers of its pair of channel ends[†]. The ends of a newly created channel are not initially connected to any component instance. Like other values, channel end identifiers can be spread within or among component instances by copying, parameter passing, or through writing/reading them to/from channel ends. This way, channel ends created in an active entity within one component instance can become known in other active entities in the same or another component instance. There is no explicit operation in Reo to delete a channel. In practice, useless channels that can no longer be referred to by any (active entity in any) component instance may be garbage collected.

3.2. *Channel* `_forget`

The `_forget` operation changes its e argument such that it no longer refers to the channel end it designates. An active entity that (indirectly) performs this operation (by performing its corresponding Reo operation `forget` described in Section 4.1.2) causes e to be forgotten by *all* active entities inside the same (immediately enclosing) component instance. This contributes to the eligibility of a channel as a candidate for garbage collection.

3.3. *Channel* `_move`

The `_move` operation moves the channel end identified by its e argument to the specified location. Although mobility of channel ends has significant consequences both for the applications as well as the implementation of channels, it is indeed transparent to Reo. The only consequence of moving a channel end is that it may allow more efficient access to the channel end and the data content of the channel by subsequent channel operations performed by the active entities at the new location. The location or moving of a channel end does not disrupt the state of or the flow of data through the channel.

[†] Earlier papers on Reo stipulated an optional *pattern* argument for `_create` to specify a *filter* for the new channel. This has now been dropped because the same effect can be obtained by `join`'ing an appropriate `Filter(`*pattern*`)` channel with any arbitrary channel. See Section 6.2.

3.4. *Channel* `_connect`

The `_connect` operation succeeds when the specified channel end is connected to the component instance that contains the active entity performing it. Pending connect requests on the same channel end are granted on a first-come-first-serve basis.

The `_connect` operation allows the same channel end to be dynamically passed around to be used by different component instances, while it preserves the one-to-one property of channel connections: at any given time, there is at most one component instance connected to each of the two ends of a channel. This way, Reo guarantees the soundness and completeness properties that are shown to be required for compositionality (Arbab et al., 2000b).

3.5. *Channel* `_disconnect`

The `_disconnect` operation succeeds when the specified channel end is disconnected from the component instance that contains the active entity performing it. Disconnecting a channel end pre-empts and retracts all `_read`, `_take`, and `_write` operations that may be pending on that channel end; as far as these operations are concerned, it is as if the channel end were not connected to the component instance in the first place. One end of a channel is oblivious to whether or not its opposite end is connected or moves.

3.6. *Channel* `_wait`

The `_wait` operation succeeds when its condition expression is true. The parameter *conds* is a boolean combination (using and, or, negation, and parentheses for grouping) of a set of predefined primitive conditions on channel ends. For our purposes in this paper, the primitive channel conditions are the ones defined in Table 2. Although the primitive conditions that appear in a wait expression may refer to different channels, a `_wait` operation preserves the atomicity of its expression: it succeeds only if the expression as a whole is true.

Condition	Description	Complement
`_connected(`e`)`	channel end e is (not) connected	`_notconnected(`e`)`
`_empty(`e`)`	channel end e is (not) empty	`_notempty(`e`)`
`_full(`e`)`	channel end e is (not) full	`_notfull(`e`)`
`_contains(`e, pat`)`	data matching *pat* does (not) exist in channel end e	`_notcontains(`e, pat`)`

Table 2. *Channel conditions*

For completeness, Reo requires the negation of every channel condition *_xxx* to also be defined as `_not`*xxx*. Thus, every condition in the first column of Table 2 has its complement condition also defined in the third column of this table. The relationship between `_notempty(e)` and `_full(e)` depends on the specific semantics of different channel types.

The `_wait` operation applies De Morgan's law on its condition expression to push

boolean negation operators all the way down to be "absorbed" by its primitive channel conditions. Thus, for instance, $\neg(\texttt{\_connected}(x) \wedge \texttt{\_full}(y))$ becomes $\neg\texttt{\_connected}(x) \vee \neg\texttt{\_full}(y)$, which allows the two negation operators to be absorbed by their respective primitive conditions, yielding the simplified "positive" condition expression $\texttt{\_notconnected}(x) \vee \texttt{\_notfull}(y)$.

3.7. *Channel* `_read`

The `_read` operation succeeds when a data item that matches with the specified pattern *pat* is available for reading through the sink channel end *inp* and it is read into the specified variable *v*. If no explicit pattern is specified, the default wild-card pattern `*` is assumed. When no variable is specified, no actual reading takes place, but the operation succeeds when a suitable data item is available for reading. Observe that the `_read` operation is non-destructive, i.e., the data item is only copied but not removed from the channel.

3.8. *Channel* `_take`

The `_take` operation is the destructive version of `_read`, i.e., the data item is actually removed from the channel. When no variable is specified as the destination in a `_take` operation, the operation succeeds when a suitable data item is available for taking and it is removed through the specified channel end.

3.9. *Channel* `_write`

The `_write` operation succeeds when the content of the specified variable is consumed by the channel to which *outp* belongs.

4. Connectors

A **connector** is a set of channel ends and their connecting channels organized in a graph of **nodes** and **edges** such that:

— Zero or more channel ends coincide on every node.
— Every channel end coincides on exactly one node.
— There is an edge between two (not necessarily distinct) nodes if and only if there is a channel whose ends coincide on those nodes.

We use $x \mapsto N$ to denote that the channel end x coincides on the node N, and $\widehat{x}$ to denote the unique node on which the channel end x coincides. For a node N, we define the set of all channel ends coincident on N as $[N] = \{x \mid x \mapsto N\}$, and disjointly partition it into the sets $Src(N)$ and $Snk(N)$, denoting the sets of source and sink channel ends that coincide on N, respectively.

Observe that nodes are *not* locations. A node is a fundamental concept in Reo representing an important topological property: all channel ends $x \in [N]$ coincide on the same node N. This property entails specific implications in Reo regarding the flow of

data among the channel ends $x \in [N]$, irrespective of concern for any location. While in practice, an implementation of Reo should try to internally `_move` all channel ends $x \in [N]$ to the same location in order to improve the efficiency of data-flow operations through a node N, strictly speaking, this is unnecessary and the semantics of a node in Reo does not require it to correspond to or reside on any specific location.

A node N is called a **source node** if $Src(N) \neq \emptyset \wedge Snk(N) = \emptyset$. Analogously, N is called a **sink node** if $Src(N) = \emptyset \wedge Snk(N) \neq \emptyset$. A node N is called a **mixed node** if $Src(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$.

The graph representing a connector is *not* directed. However, for each channel end x_c of a channel c, we use the directionality of x_c to assign a *local direction in the neighborhood of* $\widehat{x_c}$ to the edge that represents c. The local direction of the edge representing a channel c in the neighborhood of the node of its source x_c is presented as an arrow emanating from $\widehat{x_c}$. Likewise, the local direction of the edge representing a channel c in the neighborhood of the node of its sink x_c is presented as an arrow pointing to $\widehat{x_c}$.

By definition, every channel represents a (simple) connector. More complex connectors are constructed in Reo out of simpler ones using the `join` operation described in Section 4.1.7.

4.1. *Node Operations*

Table 3 shows the node counterparts of the operations in Table 1. The names of the operations in Table 3 do not have underscore prefixes: they are meant to be used by components. The operations in Table 3 that modify nodes are defined only on non-hidden nodes (see Section 4.1.9). They all fail with an appropriate error if any of their node arguments is hidden. As in Table 1, the second column in Table 3 shows whether the connectivity of (all channel ends coincident on) the node argument(s) of each operation is a prerequisite for that operation.

Observe that although syntactically the operations in Table 1 have channel end arguments, semantically they operate on nodes, *not* on channel ends. In other words, semantically, these operations are truly *node operations*, in spite of the fact that syntactically, their arguments are channels ends. A channel end e is merely a shorthand for the node $\widehat{e}$. It is convenient to use channel ends as arguments for Reo operations to indirectly designate their corresponding nodes, instead of requiring direct syntactic references to those nodes. This indirection alleviates the need for components to deal with nodes explicitly as separate entities. The components know and manipulate only channel ends. These are created by the `create` operation, and are passed as arguments to the other operations in Table 3, where they actually represent the nodes that they coincide on, rather than the specific channel ends that they are. This makes components immune to the dynamic creation and destruction of the nodes, while third parties perform `join` and `split` operations on those nodes. In effect, a node operation (e.g., `connect`, `wait`, or an I/O operation) involving a channel end e is an operation on the node $\widehat{e}$, although the actual node designated by $\widehat{e}$ may change between the time when the operation is issued and when it is eventually performed.

For example, suppose a component instance C_1 knows a channel end e, which at the

moment happens to topologically coincide on the node N_1. Because components are denied any direct means to refer to nodes, the only way C_1 can refer to N_1 is as $\widehat{e}$ (or $\widehat{e'}$ if it also knows another channel end e' that happens to currently coincide on N_1 as well). A component cannot rely on any property of $\widehat{e}$ other than the trivial property $e \in [\widehat{e}]$ (specifically, it cannot generally depend on $\widehat{e}$ and $\widehat{e'}$ to actually be the same node). C_1 is thus unaffected by any change of topology that may occur at N_1 (although not necessarily by the consequences of this change). For instance, if another component instance, C_2, performs a join or a split operation on N_1, making e coincide on another node, N_2, C_1 remains oblivious to this topological change and can still refer to $\widehat{e}$, which now designates the node N_2.

Defining Reo operations such that an operation *op* is written as $op(e)$ instead of $op(\widehat{e})$ accommodates a convenient short-hand and makes it unnecessary for components to even have a data type for nodes. However, placing nodes in a semantic domain that is beyond the realm of direct syntactic access and manipulation by components is a subtle design decision with somewhat more profound implications on the conduct of coordination.

Coordination can be conducted endogenously, or exogenously (Arbab, 1998). In *endogenous* coordination models the primitives that cause and affect the coordination of an entity with others can reside only inside of that entity itself. For instance, this is the case for models based on object oriented message passing paradigms. In *exogenous* coordination models the primitives that cause and affect the coordination of an entity with others generally reside inside of other entities. Exogenous coordination models allow third parties to orchestrate the interactions among others. An underlying exogenous coordination model is essential for a component model in which components are building blocks to be (dynamically) composed together by other entities. This is the case in Reo: topologies of connectors (i.e., coincidence of channel ends on nodes) can generally be manipulated by (active entities inside) component instances that are not (necessarily) subjects of the coordination protocols they impose. The fact that component instances have no direct access to nodes allows topological manipulations to take place without their knowledge or involvement. This facilitates exogenous coordination, making it possible to change a node without having to interfere with or update any part of the component instances that (only indirectly) refer to that node (through its coincident channel ends).

4.1.1. *Node Create* A `create` operation (1) performs its corresponding `_create` operation, (2) creates a node for each of the two new resulting channel ends, and (3) returns these same channel ends.

4.1.2. *Node Forget* A `forget` operation performed by (an active entity inside) a component instance on the node $\widehat{e}$, atomically performs the set of channel operations `_forget`(x), for all channel ends $x \in [\widehat{e}]$. When (an active entity inside) a component instance performs a `forget` operation on one of the nodes that it is connected to, all I/O operations pending on that node are retracted (i.e., they fail).

Strictly speaking, the `forget` operation itself does not directly affect the connection status of its operand node. However, `forget`'ing a connected node always makes it eligible

Operation	Con.	Description
`create(`*`chantype`*`)`	-	This operation performs `_create(`*`chantype`*`)`, creates a node for each of the two resulting channel ends, and returns the identifiers of the two channel ends.
`forget(`e`)`	N	This operation atomically performs the set of operations `_forget(`x`)`, $\forall x \in \widehat{[e]}$.
`move(`e, loc`)`	Y	This operation atomically performs the set of operations `_move(`x, loc`)`, $\forall x \in \widehat{[e]}$.
`connect(`*`[t,]`* e`)`	N	If $\widehat{e}$ is not a mixed node, this operation atomically performs the set of operations `_connect(`*`[t,]`* x`)`, $\forall x \in \widehat{[e]}$.
`disconnect(`e`)`	N	This operation atomically performs the set of operations `_disconnect(`x`)`, $\forall x \in \widehat{[e]}$.
`wait(`*`[t,] nconds`*`)`	N	This operation succeeds when the conditions specified in *nconds* become true.
`read(`*`[t,]`* e*`[, v[, pat]]`*`)`	Y	If $\widehat{e}$ is a sink node connected to the component instance, this operation succeeds when a value compatible with *pat* is non-destructively read from any one of the channel ends $x \in \widehat{[e]}$ into the variable v.
`take(`*`[t,]`* e*`[, v[, pat]]`*`)`	Y	If $\widehat{e}$ is a sink node connected to the component instance, this operation succeeds when a value compatible with *pat* is taken from any one of the channel ends $x \in \widehat{[e]}$ and read into the variable v.
`write(`*`[t,]`* e`, `v`)`	Y	If $\widehat{e}$ is a source node connected to the component instance, this operation succeeds when a copy of the value v is written to every channel end $x \in \widehat{[e]}$ atomically.
`join(`e_1`, `e_2`)`	Y	If at least one of the nodes $\widehat{e_1}$ and $\widehat{e_2}$ is connected to the component instance, this operation merges them into a new node (i.e., after the join, $\widehat{e_1}$ and $\widehat{e_2}$ become synonyms for the same node).
`split(`e*`[, quoin]`*`)`	N	This operation produces a new node N and splits the set of channel ends in $\widehat{[e]}$ between the old $\widehat{e}$ and N, according to the set of edges specified in *quoin*.
`hide(`e`)`	N	This operation hides the node $\widehat{e}$ such that it cannot be modified in any other operation.

Table 3. *Node operations*

for the implicit `disconnect` rule described in Section 4.1.10, which promptly disconnects the node.

4.1.3. *Node I/O Operations* A `read`, `take`, or `write` operation performed by (an active entity inside) a component instance on a channel end e becomes and remains *pending* on the node $\widehat{e}$ (although the actual node $\widehat{e}$ may change while the operation is pending, as described in Sections 4.1.7 and 4.1.8) until either its time-out expires, or the conditions are right for the execution of its corresponding channel end operation(s). The node I/O operations can succeed only if the nodes they refer to are connected to the component instances that (contain the active entities that) perform them. Because mixed nodes cannot be connected to any component instance (see Sections 4.1.5 and 4.1.7), `read`, `take`, and `write` cannot be performed on mixed nodes.

The precise semantics of `read`, `take`, and `write`, as well as the semantics of mixed

nodes, depends on the generic properties of the channels that coincide on their involved nodes. This is described in Section 8.

Intuitively, `read` and `take` operations nondeterministically obtain one of the suitable values available from the sink channel ends that coincide on their respective nodes. The `write` operation, on the other hand, replicates its value and atomically writes a copy to every source channel end that coincides on the node of its channel-end parameter.

A component instance C may include more than one active entity. Each of such active entity can concurrently issue an I/O operation on a node N connected to C, specifying the same or different channel ends in $[N]$. Reo defines the semantics of concurrent node I/O operations consistently with the semantics of its mixed nodes. When multiple `write` operations are concurrently pending on a source node N, they can succeed only one at a time: whenever a value can be replicated to all source channel ends in $[N]$, a pending `write` operation is selected to succeed nondeterministically. When multiple `take` operations are concurrently pending on a sink node N, they can succeed only simultaneously: any nondeterministically selected value available through one of the sink channel ends in $[N]$ that matches with the read-patterns of all pending `take` operations is removed from its channel end and replicated to all pending `take` operations to make them succeed.

4.1.4. *Node Move* The `move` operation in Reo exists only to accommodate efficient performance in distributed systems: it enables physical relocation of channel ends, but entails no semantic consequences. As defined in Table 3, the `move` operation performed on a channel end e atomically moves all channel ends that coincide on $\widehat{e}$ to its location argument, *loc*. This may allow more efficient access to these channel ends by subsequent operations performed at the specified location.

There are three occasions where moving a node may be useful. First, when a component instance connects to a node, it typically intends to subsequently perform some I/O operations on that node. Thus, often a `move` operation immediately follows a `connect`. Second, when a component instance moves from one location to another, all of its currently connected nodes should also move together with it to preserve the efficiency of its subsequent channel end operations at its new location. In this case, the (non-Reo) component-instance-move operation should perform the respective node `move` operations as well. Third, a distributed component instance may move a node to a location in order to allow more efficient operations on that node by its internal active entities.

4.1.5. *Node Connection* As defined in Table 3, the `connect` and `disconnect` operations performed on a channel end e atomically connect and disconnect all channel ends that coincide on $\widehat{e}$ to their respective component instances. Only source and sink nodes (not mixed nodes) can be connected to component instances. Thus, a `connect` fails if the node of its argument channel end is a mixed node.

When a node is disconnected from a component instance, all `read`, `take`, and `write` operations pending on that node are pre-empted and retracted; as far as these operations are concerned, it is as if the node were not connected to the component instance in the first place when those operations were attempted.

The only way in Reo to connect a node N to a component instance C is through

(an active entity inside) C performing a `connect` operation on N. There are three ways in Reo in which a node N connected to a component instance C can be disconnected: (1) an active entity inside C performs an explicit `disconnect` operation on N; (2) an active entity inside C performs a `join` operation that relinquishes its connection to N (Section 4.1.7); and (3) the node N becomes eligible for the implicit disconnect rule (Section 4.1.10).

4.1.6. *Node Conditions* The *nconds* in `wait` is a boolean combination of primitive node conditions, which are the counterparts of the primitive channel end conditions of `_wait` in Table 1. For every primitive condition *_xxx(e)* on a channel end e, Reo defines two corresponding primitive conditions on its node $\widehat{e}$: *xxx(e)* and *xxx*`All`*(e)*. A primitive node condition without the `All` suffix is true for a node $\widehat{e}$ when its corresponding channel end condition is true for some channel end $x \in [\widehat{e}]$. Analogously, a primitive node condition that ends with the suffix `All`, is true for a node $\widehat{e}$ if its corresponding channel end condition is true for all channel ends $x \in [\widehat{e}]$. Table 4 summarizes the node conditions corresponding to the channel end conditions of Table 1.

Condition	Description	Complement
`connected(`e`)`	some channel end in $[\widehat{e}]$ is (not) connected	`notconnected(`e`)`
`connectedAll(`e`)`	every channel end in $[\widehat{e}]$ is (not) connected	`notconnectedAll(`e`)`
`empty(`e`)`	some channel end in $[\widehat{e}]$ is (not) empty	`notempty(`e`)`
`emptyAll(`e`)`	every channel end in $[\widehat{e}]$ is (not) empty	`notemptyAll(`e`)`
`full(`e`)`	some channel end in $[\widehat{e}]$ is (not) full	`notfull(`e`)`
`fullAll(`e`)`	every channel end in $[\widehat{e}]$ is (not) full	`notfullAll(`e`)`
`contains(`e, pat`)`	some channel end in $[\widehat{e}]$ does (not) contain data matching *pat*	`notcontains(`e, pat`)`
`containsAll(`e, pat`)`	every channel end in $[\widehat{e}]$ does (not) contain data matching *pat*	`notcontainsAll(`e, pat`)`

Table 4. *Node conditions*

Note the precedence of `not`, which is applicable at the channel end level, over `All`, which applies at the node level: the condition `notemptyAll(`e`)`, for instance, is true if all channel ends that coincide on the node $\widehat{e}$ are non-empty. The situation where not all channel ends coincident on a node $\widehat{e}$ are empty can be expressed as the condition `notempty(`e`)`, which holds if there exists at least one channel end coincident on $\widehat{e}$ that is non-empty.

A `wait` operation thus translates its node condition expression into a channel end condition expression, and uses it to perform a `_wait` operation. Observe that the above syntax rules for deriving node condition names from channel-end condition names yield node condition names `connected(`e`)` and `connectedAll(`e`)` from the channel-end condition name `_connected(`e`)`, and node conditions `notconnected(`e`)` and `notconnectedAll(`e`)` from the channel-end condition `_notconnected(`e`)`. On the other hand, the semantics of the `connect` operation guarantees that `connected(`e`)` $\Longrightarrow$ `connectedAll(`e`)` and

`notconnected(`e`)` $\Longrightarrow$ `notconnectedAll(`e`)`, respectively making each a synonym for the other.

4.1.7. *Node Composition* The composition operation `join(`e_1`,` e_2`)` succeeds only if at least one of the two nodes $\widehat{e_1}$ and $\widehat{e_2}$ is connected to the component instance, p, containing the active entity that performs this operation. The effect of `join` is the merge of the two nodes $\widehat{e_1}$ and $\widehat{e_2}$ into a new common node, N; i.e., for all channel ends $x \in [\widehat{e_1}] \cup [\widehat{e_2}]$, we have $\widehat{x} = N$.

Types of $\widehat{e_1}$ and $\widehat{e_2}$	Connection of the other node	Connection status of the result
different	connected to p	not connected
	not connected	not connected
	connected to $q \neq p$	`join` fails
same	connected to p	connected to p
	not connected	not connected
	connected to $q \neq p$	connected to q

Table 5. *p performs* `join(`e_1, e_2`)` *while at least one of the nodes $\widehat{e_1}$, $\widehat{e_2}$ is connected to p*

Table 5 summarizes the rules for node composition when an active entity inside a component instance p performs a `join(`e_1`,` e_2`)`. This operation can succeed only if at least one of the two nodes $\widehat{e_1}$ and $\widehat{e_2}$ is connected to p. The other node may be connected to p, not connected to any component instance, or connected to another component instance, q, distinct from p. Depending on the connection status of the other node, and whether or not the two nodes are of the same type, the `join` operation may fail or succeed, and the resulting node may or may not remain connected to p. Observe that `join` never affects the connectivity of a component instance other than p with any node. If the two nodes are of different types, then the resulting merged node would be a mixed node, which cannot be connected to any component instance. Table 5 shows that this constraint is preserved because in this case either p relinquishes its connection, or the `join` operation fails, when this is not enough.

The rationale for having connection to a node as a prerequisite for the `join` operation is to ensure a simple, deterministic, local contention resolution scheme in distributed systems. If two component instances attempt composition operations at the same time, conflicts and/or race conditions may arise. If at least one of the two nodes in a composition is connected to the component instance that attempts the composition, the conflict can be resolved by requiring it to relinquish its (exclusive) connection. The rationale for not requiring both nodes to be connected to the component instance that attempts the `join` is to avoid crippling the conduct of exogenous coordination. If $\widehat{e_2}$ is connected to a component instance q, assuming that it is not a hidden node (see Section 4.1.9), an active entity in another component instance, p, should be able to modify the topology of $\widehat{e_2}$, e.g., by merging it with $\widehat{e_1}$, without the involvement or the knowledge of q. This would not be possible if to perform `join`, p were required to be connected to both nodes.

When a component instance p loses its previous connection with any of the nodes $\widehat{e_1}$ and $\widehat{e_2}$, all `read`, `take`, and `write` operations pending on that node are retracted; i.e., they fail as if their respective nodes were not connected to the component instance in the first place. Otherwise, all operations pending on $\widehat{e_1}$ and $\widehat{e_2}$ remain pending on their common heir, N. Specifically, observe that if, e.g., $\widehat{e_1}$ is connected to p and $\widehat{e_2}$ is connected to another component instance, q, then a `join(`e_1`,` e_2`)` performed by an active entity inside p does *not* disrupt any operation (issued by an active entity inside q) that may be pending on $\widehat{e_2}$. See Section 8 for the semantics of mixed nodes and the semantics of I/O operations on other nodes.

4.1.8. *Node Splitting* A `split` operation performed on a node $N = \widehat{e}$ produces a new node N' and divides the set of channel ends $[N]$ between the two nodes N and N'. The `split` operation does not require its node argument, $\widehat{e}$, to be connected to the component instance (that contains the active entity) that performs it. Furthermore, strictly speaking, it does not directly affect the connection of $\widehat{e}$ to any component instance that it may be connected to. However, the node $\widehat{e}$ may become eligible for the implicit disconnect rule after the `split` operation (see Section 4.1.10). The newly created node, N', initially shares the same connection status as that of $\widehat{e}$ before the `split`. However, it too can become eligible for the implicit disconnect rule after the `split` operation. Consequently, any I/O operation that may be pending on $\widehat{e}$ before the split, remains unaffected and pending on either N or N' after the split.

Different versions of the `split` operation, with different signatures, allow different ways of specifying how the channel ends in $[N]$ are to be split between N and N'. One way or the other, the ends of the channels that form the "exterior angle" at the splitting node constitute the **quoin** of the split operation, and they are the ones that are moved to the new node.

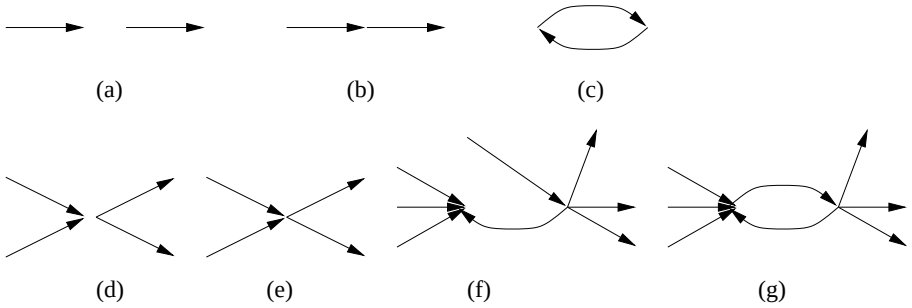

Fig. 3. Examples of join and split

In `split(`e`,` S`)`, the parameter S is a set of channel ends and every channel end $x \in [\widehat{e}] \cap S$ is moved to the new node, N'. The operation `split(`e`)` moves all sink channel ends $x \in \mathit{Snk}(\widehat{e})$ to the new node N', leaving only source channel ends to coincide on N. The quoin of the split in `split(`e_1`,` e_2`)` is defined through the set Q of channels with one end on each of the two nodes $\widehat{e_1}$ and $\widehat{e_2}$: the ends of the channels in Q that are in $[\widehat{e_1}]$ are moved to N'.

Figure 3 shows a few examples of `join` and `split` operations. By joining the sink and the source ends of the two channels in Figure 3.a, we obtain the connector in Figure 3.b. A split performed on the mixed node in Figure 3.b inverses the join operation and produces the two independent channels of Figure 3.a. Joining the source and the sink nodes of the connector in Figure 3.b, produces the connector in Figure 3.c. Similarly, the connector in Figure 3.b can be obtained by splitting one of the mixed nodes of the connector in Figure 3.c. Analogously, the pairs of Figures 3.d and e and f and g show connectors that are related to each other through a one-node `join` and `split` operations, respectively.

4.1.9. *Hiding Nodes* The `hide(`e`)` operation is an important abstraction mechanism in Reo: hiding a node $N = \widehat{e}$ ensures that the topology of N can no longer be modified in any other operation (by any active entity in any component instance). Any operation that may entail a change to $\widehat{x}$ for any $x \in [N]$ fails after this `hide` operation. This guarantees that the topology of channels coincident on N can no longer be modified by anyone. Observe that hidden nodes can still be used in Reo operations (e.g., for I/O) that do not modify nodes.

4.1.10. *Implicit Disconnection* The implicit disconnection rule in Reo states that a component instance C cannot remain connected to a node N if no active entity in C *knows* a channel end $e \in [N]$. If N is not connected to C and no active entity in C knows any channel end in $[N]$, then N cannot *become* connected to C because, clearly, no `connect` operation can be performed to connect N to C. However, there are situations when C loses its reference to (i.e., its knowledge of) a node N to which it is already connected. The implicit disconnection rule ensures that in such cases N is disconnected from C, because otherwise N would remain connected for good to a component instance which has no way to refer to it, while other component instances may be waiting to connect to and use this node.

A simple case of the application of this rule is when (an active entity inside) C performs a `forget` on a node N that it is already connected to. After the `forget` operation, there is no way for any entity inside C to refer to N. Losing its last reference to (channel ends in) N triggers an implicit `disconnect` operation on behalf of C on this node.

A more subtle case of the application of the implicit disconnection rule is when a `split` operation performed on a connected node produces a connected node none of whose coincident channel ends is known to the component instance it is connected to. For example, suppose a component instance, C, is connected to a node N where $[N] = \{a, b, c\}$ out of which C knows only the channel end a. If a `split` operation (perhaps performed by a third party) splits N into N_1 and N_2 such that $[N_1] = \{a\}$ and $[N_2] = \{b, c\}$, C cannot remain connected to N_2 because it has no reference to this node.

5. Patterns

Reo uses patterns to regulate channel input/output operations. A **pattern** is an expression that matches a data item when it is written to, read from, or simply flows through a channel. The operations `take` and `read` can specify patterns that must match the

items they read. Furthermore, some channel types may require patterns as their creation parameters that influence their behavior.

We write $d \ni p$ to denote that the data item d matches with the pattern p, and $d \not\ni p$ to denote otherwise. The precise syntax of patterns is not important in this paper. We illustrate the utility of patterns with some examples.

The atomic patterns are type identifiers (e.g., `int`, `real`, `string`, `number`, etc.) that match with any one of their instances, plus the wild-card pattern (*). A specific value is a pattern that matches only itself. Patterns can be composed into tuple structures using angular brackets (`<` and `>`). Thus, `<int, string>` is a pattern that matches any pair that consists of an integer and a string. Matched patterns can bind free variables, which in turn can be used to enforce additional constraints. For instance, `<int*x, 2.4, x>` matches any triplet consisting of the same integer as its first and third element, with the `real` value 2.4 as its second.

A pattern can be augmented with additional constraints in square brackets. For instance, `<int*x, *, int*y>[x > y]` matches with any triplet with two integers as its first and third elements, as long as the first element is numerically greater than the third. The pattern `<int*x, string[a+b*c], real*y> [y >= 3*x]` matches triplets consisting of an integer, a string, and a real number, where the real number is greater than or equal to 3 times the integer, and the string consists of one or more occurrences of "a" followed by zero or more occurrences of "b" with a single "c" at its end.

6. Channel Types

Reo assumes the availability of an arbitrary set of channel types, each with its well-defined behavior. A channel is called **synchronous** if it delays the success of the appropriate pairs of operations on its two ends such that they can succeed only simultaneously; otherwise, it is called **asynchronous**. An asynchronous channel may have a bounded or an unbounded buffer (to hold the data items it has already consumed through its source, but not yet dispensed through its sink) and may or may not impose a certain order on the delivery of its contents. A **lossy** channel may deliver only some of the data items that it receives, and lose the rest.

Although every channel in Reo has exactly two ends, they may or may not be of different types. Thus, a channel may have a source and a sink end, two source ends, or two sink ends. The behavior of a channel may depend on such parameters as its synchronizing properties, the number of its source and sink ends, the size of its buffer, its ordering scheme, its loss policy, etc.

While Reo assumes no particular fixed set of channel types, it is reasonable to expect that a certain number of commonly used channel types will be available in all implementations and applications of Reo. Tables 6 and 7 show a non-exhaustive set of interesting channel types and their properties. A larger set of channel types are described in (Arbab, 2002). Most of these channel types are indicative examples only. The few that appear in Tables 6 and 7 are those that are used further in this paper as the building blocks for more complex connectors to demonstrate the expressiveness of Reo.

Type	Description
`Sync`	has a source and a sink. The pair of I/O operations on its two ends can succeed only simultaneously.
`Filter(`*pat*`)`	has a source and a sink. Writing a data item that does not match with the specified pattern *pat* always succeeds immediately and the data item is lost. For data items that match the pattern *pat* this channel behaves the same way as a `Sync` channel.
`SyncDrain`	has two source ends. The pair of I/O operations on its two ends can succeed only simultaneously. All data items written to this channel are lost.
`SyncSpout(`*pat*`)`	has two sink ends. The pair of I/O operations on its two ends can succeed only simultaneously. Each sink of this channel acts as an unbounded source of data items that match with the pattern *pat*. Data items are produced in a nondeterministic order. The data items taken out of the two sinks of this channel are not related to each other.
`LossySync`	has a source and a sink. The source end always accepts all data items. If there is no matching I/O operation on the sink end of the channel at the time that a data item is accepted, then the data item is lost; otherwise, the channel transfers the data item exactly the same as a `Sync` channel, and the I/O operation at the sink end succeeds.

Table 6. *Examples of synchronous channel types*

6.1. *Channel Type* `Sync`

The `Sync` channel type represents the typical synchronous channels. A `_read`(t, y_c, v, p) on the sink y_c of a channel c of this type succeeds only if there is a `_write`(t', x_c, d) operation pending on the source x_c of this channel and the data item d matches the read-pattern p. In this case, d is copied into the read-variable v, the `_read` operation succeeds, but the `_write` remains pending.

A `_write`(t', x_c, d) operation succeeds only if there is a `_take`(t, y_c, v, p) operation pending on the sink y_c of the channel c, and the data item d matches the take-pattern p. In that case, d is copied into the read-variable v, and the `_take` operation succeeds simultaneously as well.

6.2. *Channel Type* `Filter(`*pat*`)`

A `Filter(`*pat*`)` channel type is a special lossy synchronous channel. It transfers only those data items that match with its specified filter pattern *pat* and loses the rest. A `_read`(t, y_c, v, p) on the sink y_c of a channel c of this type succeeds only if there is a `_write`(t', x_c, d) operation pending on the source x_c of this channel and the data item d matches both with the filter pattern *pat* as well as the read-pattern p. In this case, d is copied into the read-variable v, the `_read` operation succeeds, but the `_write` remains pending.

A `_write`(t', x_c, d) operation succeeds only if either (1) d does not match with the filter pattern *pat*; or (2) there is a `_take`(t, y_c, v, p) operation pending on the sink y_c of the channel c, and the data item d matches both with the filter pattern *pat* as well as the take-pattern p. In the former case, the data item d is lost. In the latter case, d is copied into the read-variable v, and the `_take` operation succeeds simultaneously as well.

Type	Description
`FIFO`	has a source and a sink, and an unbounded buffer. The source end always accepts all data items. The accepted data items are kept in the internal FIFO buffer of the channel. The appropriate operations on the sink end of the channel obtain the contents of the buffer in the FIFO order.
`FIFO`n	is the bounded version of `FIFO` with the channel buffer capacity of n data items.
`AsyncDrain`	has two source ends. The channel guarantees that two operations on its two ends never succeed simultaneously. The channel is *fair* by alternating between its two ends and giving each a chance to dispose of a data item. All data items written to this channel are lost.
`AsyncSpout(`*pat*`)`	has two sink ends. The channel guarantees that two operations on its two ends never succeed simultaneously. The channel is *fair* by alternating between its two ends and giving each a chance to obtain a data item from the channel. The values obtained from the two ends of the channel are not related to each other, but match with the specified pattern *pat*.
`ShiftFIFO`n	is the lossy version of `FIFO`n, where the arrival of a data item when the channel buffer is full, triggers the loss of the oldest data item in the buffer, to make room for the new arrival.
`LossyFIFO`n	is the lossy version of `FIFO`n, where all newly arrived data items when the channel buffer is full, are lost.

Table 7. *Examples of asynchronous channel types*

6.3. *Channel Type* `SyncDrain`

A `SyncDrain` is a lossy channel that allows pairs of `_write` operations pending on its opposite ends to succeed simultaneously, thus, synchronizing them. All written values are lost.

6.4. *Channel Type* `SyncSpout`

A `SyncSpout(`*pat*`)` channel is an unbounded source of data items that match with its specified pattern, *pat*, and can be taken from its opposite ends only simultaneously in some nondeterministic order. While the pair of `_take` operations performed on the opposite ends of a `SyncSpout(`*pat*`)` are synchronized by the channel, the two data items taken by these operations are independent of each other. For example, `<x, y> = create(SyncSpout(int*x[0 <= x, x <= 10]))` creates a `SyncSpout` channel each of whose two sink ends `x` and `y` produces an unbounded sequence of integers between 0 and 10 in some nondeterministic order. Read operations on `x` and `y` succeed immediately independent of each other and successive read operations on the same end, of course, produce the same integer (read is non-destructive). However, a take operation on one end can succeed only simultaneously with another take operation at the other end.

6.5. *Channel Types* `FIFO` *and* `FIFO`n

The `FIFO`, and `FIFO`n channel types, where n is an integer greater than zero, represent the typical unbounded asynchronous and bounded asynchronous FIFO channels. A `_write` to a `FIFO` channel always succeeds, and a `_write` to a `FIFO`n channel succeeds only if

the number of data items in its buffer is less than its bounded capacity, n. A `_read` or _take from a `FIFO` or `FIFO`n channel suspends until the first (i.e., oldest) data item in the channel buffer matches with the `_read` or _take pattern, in which case, it is (destructively) obtained and the operation succeeds.

6.6. *Channel Types* `AsyncDrain` *and* `AsyncSpout`

`AsyncDrain` and `AsyncSpout(`*pat*`)` are analogous to `SyncDrain` and `SyncSpout(`*pat*`)`, respectively, except that they guarantee that, respectively, the pairs of `_write` and the pairs of `_take` operations on their opposite ends never succeed simultaneously.[‡] These channel types are important basic synchronization building blocks for the construction of more complex connectors.

6.7. *Lossy Channels*

An important class of channel types is the so-called lossy channels. These are the channels that do not necessarily deliver through their sinks every data item that they consume through their sources. For instance, `SyncDrain` and `AsyncDrain` channels are lossy channels that lose every data item written to them. `Filter(`*pat*`)` is a lossy channel that passes only those data items that match its specified pattern *pat* and loses the rest.

A channel can be lossy because when its bounded capacity becomes full, it follows a policy to, e.g., drop the new arrivals (overflow policy) or the oldest arrivals (shift policy). `ShiftFIFO`n is a bounded capacity FIFO channel that loses the oldest data item in its buffer when its capacity is full and a new data item is to be written to the channel. Thus, (up to) the last n arrived data items are kept in its channel buffer. A `LossyFIFO`n channel, on the other hand, loses the newly arrived data items when its capacity is full.

An asynchronous channel may be lossy because it requires an expiration date for every data item it consumes, and loses any data item that remains in its buffer beyond its expiration date. Other channels may be lossy because they implement other policies to drop some of the data items they consume.

A `LossySync` channel behaves the same as a `Sync` channel, except that a write operation on its source always succeeds immediately. If a compatible read or take operation is already pending on the sink of a `LossySync` channel, then the written data item is transferred to the pending operation and both succeed. Otherwise, the write operation succeeds and the data item is lost.

[‡] Excluding the possibility of simultaneous success of the operations at the two ends of these channels may seem more rigid than what the usual connotation of the term “asynchronous” implies. However, observe that there is nothing sacrosanct about these or any other channel names or definitions in Reo, and users can (re)define their own sets of channels as they please. These specific channels, as defined here, simply turn out to be useful, e.g., in the connector of Figure 5.f.

7. Channel Behavior

The channel types described in Section 6 are indicative of the richness and the diversity of the behavior of channels allowed in Reo. However, Reo is not directly aware of the behavior of any particular channel. Reo expects every channel type to be able to provide a "reasonable implementation" of the operations in Table 1. We state "reasonable implementation" rather than imposing a rigid semantics here, because we do not wish to preclude such reasonable possibilities as, for instance, a "read-only" channel type for which `take` becomes a synonym for `read`.

The set of operations in Table 1, thus, describes the **common behavior** of all channels in Reo. However, the operations in Table 1 are not sufficient to describe the full behavior of different channel types. As far as Reo is concerned, the **generic behavior** of a channel c, whose source and sink are x_c and y_c, respectively, is defined indirectly through the following (state-dependent) functions:[§]

— `offers`(y_c, p) is the multi-set of pairs $\langle y_c, d\rangle$ for each d in the multi-set of values that may be assigned to a variable v in a `_take`$(0, y_c, v, p)$.
— `accepts`(x_c, d) is true for a data item d if the state of c allows `_write`$(0, x_c, d)$ to succeed.

For completeness, we define `offers`$(x_c, p) = \emptyset$, for all patterns p, and `accepts`$(y_c, d) =$ *false*, for all data items d.

In addition to its common and generic behavior, each channel type also has a *specific behavior*. The **specific behavior** of a channel type is the precise semantics that relates its generic behavior, its common behavior, and its internal state. Although the specific behavior of a channel is important wherever it is used, the Reo operations are semantically independent of the specific behavior of channels. Reo and its operations depend only on the returned results of the functions `offers()`, and `accepts()`, which under Reo's interpretation, above, comprise the generic behavior of channels. The actual definitions of these functions which relate them to the internal states and other specific details of various channels constitute their specific behavior.

8. Dataflow Through Nodes

The (active entities inside) component instances can write data values to source nodes and read/take them from sink nodes, using the node operations defined in Table 3. Generally, everything flows in Reo from source nodes through channels and mixed nodes down to sink nodes. Some data items get lost in the flow, and some settle as sediments in certain channels for a while before they flow through, if ever. It is the composition of channels into connectors, together with the node operations read, take, and write, that yield this intuitive behavior in Reo. In this section, we informally describe the operational semantics of mixed nodes and the read, take, and write operations on nodes. Our exposition is

[§] The simplified versions of these functions and equations 1, 2, and 3 presented here suffice for our illustrative purposes in this paper. These functions become more complicated to accommodate topologies that include closed loops of synchronous channels, etc.

intended to show the fundamentals of a truly distributed implementation of Reo. We ignore certain aspects of timing and all locking issues here to simplify our presentation.

We use the predicate $\Pi(O)$ to designate whether or not the operation O is pending (on its respective node). From the point of view of an entity that is about to write a data item d to a node it may be useful to know if the write operation will suspend or succeed immediately. For a node N and a data item d, we define:

$$\texttt{accepts}(N, d) = \begin{cases} \bigwedge_{p \,:\, \Pi(\texttt{take}(t,N,v,p))} d \ni p & \text{if } N \text{ is a sink node} \\ \bigwedge_{x \in Src(N)} \texttt{accepts}(x, d) & \text{otherwise} \end{cases} \tag{1}$$

Equation 1 states that if N is a sink node, it accepts a data item d only if d matches with the read-patterns p of all `take` operations currently pending on N (observe that in this case, there are no source channel ends in $[N]$). Otherwise, N accepts d only if all source channel ends in $[N]$ accept d.

The semantics of `write`'ing a data item d to a node N with a time-out of $0 \leq t \leq \infty$ can now be defined as follows.

Definition 1. A write operation $\texttt{write}(t, N, d)$ remains pending on the node N, until either (1) its time-out t expires, in which case the `write` operation fails; or (2) the predicate $\texttt{accepts}(N, d)$ is true, and the set of operations $\{\texttt{\_write}(\infty, x, d) \mid x \in Src(N)\}$ atomically succeeds, in which case the `write` operation succeeds.

Observe that a `_write` operation in Definition 1 is performed only if $\texttt{accepts}(x, d)$ is true for all channel ends $x \in Src(N)$. Channels' compliance with Reo's interpretation of the `accepts` predicate (Section 7) implies that every such `_write` operation immediately succeeds.

From the point of view of an entity that is about to take a data item from a node it may be useful to know the multi-set of data items available through that node. For a node N and a pattern p, we define:

$$\texttt{offers}(N, p) = \begin{cases} \biguplus_{d \,:\, \Pi(\texttt{write}(t,N,d))} \{\langle \epsilon, d \rangle \mid d \ni p\} & \text{if } N \text{ is a source node} \\ \biguplus_{x \in Snk(N)} \texttt{offers}(x, p) & \text{otherwise} \end{cases} \tag{2}$$

Analogous to the `offers` function for the channel ends (Section 7), this function too returns a multi-set of pairs each containing a data item that matches with the pattern p, together with its producing channel end. The special symbol ϵ represents "no channel end" and marks the data items obtained directly from the `write` operations pending on the node.

According to this definition, a source node offers only the multi-set of values proposed by the write operations pending on that node that match the specified pattern p. If N is not a source node, it is either a mixed node or a sink node. A mixed node cannot be involved in any write operation in Reo. Therefore, equation 2 defines the multi-set of the values offered by a mixed or a sink node to be the (multi-set) union of all values offered by all of its coincident sinks.

The semantics of `take`'ing a value that matches with a pattern p from a node N into a variable v before the time-out $0 \leq t \leq \infty$, can now be defined as follows.

Definition 2. A take operation $\texttt{take}(t, N, v, p)$ remains pending on the node N, until either (1) its time-out t expires, in which case the `take` operation fails; or (2) $\exists\langle y, d\rangle \in \texttt{offers}(N, p)$ and the operation $\texttt{\_take}(\infty, x, v, d)$ succeeds on a nondeterministically (but fairly) selected channel end $x \in [N]$ such that $\langle y, d\rangle \in \texttt{offers}(x, p)$, in which case the `take` operation succeeds.

Observe that a `_take` operation is performed in Definition 2 only on a channel end x for which $\texttt{offers}(x, p)$ contains an appropriately matching data item. Compliance with Reo's interpretation of the `offers` predicate (Section 7) implies that such a `_take` operation succeeds in finite time.

Semantically, a $\texttt{read}(t, N, v, p)$ operation is similar to $\texttt{take}(t, N, v, p)$, but we skip its details in this paper.

Because mixed nodes cannot be connected to components, the possibility of having a mixed node involved in a `read`, `take`, or `write` operation is precluded. A mixed node automatically transfers all eligible data items from its coincident sinks to its coincident sources. The multi-set $\tau(N)$ of the pairs representing the data items that are eligible for transfer at a mixed node N is defined as

$$\tau(N) = \{\langle y, d\rangle \mid \langle y, d\rangle \in \texttt{offers}(N, *) \wedge \texttt{accepts}(N, d)\}. \tag{3}$$

Definition 3. The semantics of a mixed node N in Reo is defined as the execution of the infinite loop in Table 8 by an independent process dedicated to N. The actions in each iteration of the for-loop on line 3, starting with the selection of a $\langle y, d\rangle \in \tau(N)$, are performed atomically.

Observe that the contents of $\tau(N)$ may change between the two lines 2 and 3, and, of course, from one iteration of the for-loop on line 3 to the next. However, once a $\langle y, d\rangle \in \tau(N)$ is selected on line 3, all operations in the iteration of the loop (up to line 8) are performed atomically. This means that the channels whose ends coincide on N are properly locked for the duration of each iteration to ensure that their states do not change by any action other than those in that iteration.

Furthermore, note that the `_take` on line 4 specifies the pre-selected data item d as its take-pattern, which can match only with d. Observe that d is selected on line 3 such that $\langle y, d\rangle \in \texttt{offers}(y, *)$, which guarantees that (1) the `_take` operation on line 4 succeeds in finite time; and (2) the `_take` operation on line 4 indeed takes the value d out of the channel y (and assigns it to the take-variable v). Moreover, because $\texttt{accepts}(x, d)$ is true, the $\texttt{\_write}(\infty, x, d)$ operation on line 6 succeeds in finite time.

When a mixed node is created by a `join` or `split` operation, a new dedicated process is created to reify its semantics. Analogously, when a mixed node is destroyed by a `join` or `split` operation, its corresponding dedicated process is destroyed.

The behavior of a source node N is analogous to that of a mixed node represented by the loop in Table 8, except that instead of the `_take` operation on line 4, it releases a $\texttt{write}(t, N, d)$ operation pending on N to succeed. The behavior of a sink node N is also described by the same loop as in Table 8, except that the lines 5-7 are replaced by a loop that releases all $\texttt{take}(t, N, v, p)$ operations pending on N to succeed.

```
1   while (true) do
2     suspend until τ(N) is non-empty
3     for each ⟨y, d⟩ ∈ τ(N) do
4       _take(∞, y, v, d)
5       for each x ∈ Src(N) do
6         _write(∞, x, d)
7       done
8     done
9   done
```

Table 8. *Semantics of a mixed node*

9. Generic Behavior of Channels

It is instructive to consider a few common channel types as examples to see how their generic behavior in Reo can be defined in terms of their specific behavior. In this section, we describe the behavior of some of the channels in Tables 6 and 7.

9.1. *Generic Behavior of Asynchronous Channels*

The existence of buffers in asynchronous channels means that the behavior of one end of an asynchronous channel is decoupled from that of its other end and, instead, depends on its buffer. This makes the behavior of asynchronous channels simpler to describe. For instance, the behavior of some of the channels presented in Table 7 is described in the rest of this section.

9.1.1. *Generic Behavior of* `FIFO` Consider a `FIFO` channel c (as described in Table 7) and let x_c and y_c be its source and sink ends, respectively. Let the sequence $B(c) = \langle B_k, B_{k-1}, ...B_2, B_1\rangle$ represent the buffer of the channel c, where B_1 is its oldest element. The generic behavior of c is defined by the following two functions.

$$\texttt{offers}(y_c, p) = \begin{cases} \{\langle y_c, B_1\rangle\} & \text{if } B(c) \neq \langle\rangle \wedge B_1 \ni p \\ \emptyset & \text{otherwise} \end{cases} \tag{4}$$

$$\texttt{accepts}(x_c, d) = \mathit{true} \tag{5}$$

This states that what the sink end of c offers (for reading or taking) is the empty set if the buffer of c is empty, and a singleton (containing the first data element to be taken), otherwise. Observe that if the first element in the buffer does not match the specified pattern, the sink of a `FIFO` channel offers no value.

9.1.2. *Generic Behavior of* `FIFO`n The behavior of a `FIFO`n channel is identical to that of a `FIFO`, except that its bounded capacity may prevent it from accepting values when its bounded capacity is full. Let c be a `FIFO`n channel with $B(c) = \langle B_k, B_{k-1}, ...B_2, B_1\rangle$ representing its buffer, as in Section 9.1.1. Clearly, the constraint $|B(c)| \leq n$ must be maintained by this channel, where $|\alpha|$ represents the length of the sequence α. The generic

behavior of c is defined by the following two functions.

$$\texttt{offers}(y_c, p) = \begin{cases} \{\langle y_c, B_1\rangle\} & \text{if } B(c) \neq \langle\rangle \wedge B_1 \ni p \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

$$\texttt{accepts}(x_c, d) = |B(c)| < n \tag{7}$$

This states that $\texttt{accepts}(x_c, d)$ succeeds as long as the number of data items in the (bounded) buffer of c is less than its capacity, n.

9.1.3. *Generic Behavior of* `ShiftFIFO`n *and* `LossyFIFO`n The generic behavior of channel types `ShiftFIFO`n and `LossyFIFO`n is identical to that of a `FIFO` channel: the fact that they may lose their contents when their capacity is full, and the different policies they use to determine which data item to lose, are all part of the details of their specific behavior. As far as the Reo operations are concerned, these channel types behave as if they were `FIFO` channels.

9.2. *Generic Behavior of Synchronous Channels*

The generic behavior of synchronous channels can be defined in terms of the properties of the nodes on which their ends coincide. For instance, we define the behavior of some of the channels presented in Table 6.

9.2.1. *Generic Behavior of* `Sync` The generic behavior of a `Sync` channel c whose source and the sink ends are, respectively, x_c and y_c, is defined by the following two functions.

$$\texttt{accepts}(x_c, d) = \texttt{accepts}(\widehat{y_c}, d) \tag{8}$$

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid \langle z, d\rangle \in \texttt{offers}(\widehat{x_c}, p)\}. \tag{9}$$

9.2.2. *Generic Behavior of* `Filter(`*pat*`)` The generic behavior of a `Filter(`*pat*`)` channel c whose source and the sink ends are, respectively, x_c and y_c, is defined by the following two functions.

$$\texttt{accepts}(x_c, d) = d \not\ni pat \vee \texttt{accepts}(\widehat{y_c}, d) \tag{10}$$

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid \langle z, d\rangle \in \texttt{offers}(\widehat{x_c}, p) \wedge d \ni pat\}. \tag{11}$$

9.2.3. *Generic Behavior of* `LossySync` The generic behavior of a `LossySync` c whose source and sink ends are, respectively, x_c and y_c, is defined by the following two functions.

$$\texttt{accepts}(x_c, d) = \mathit{true} \tag{12}$$

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid \langle z, d\rangle \in \texttt{offers}(\widehat{x_c}, p)\}. \tag{13}$$

This reflects the fact that the state of a `LossySync` channel allows it to consume a data item regardless of whether or not a matching I/O operation is pending on its opposite end, and either transfers or loses the data item.

10. Channel Composition

The utility of channel composition in Reo can be demonstrated through a number of simple examples. For convenience, we represent a channel by the pair of its source and sink ends, i.e., `ab` represents the channel whose source and sink ends are, respectively, `a`, and `b`. Two channels, `ab` and `cd` can be joined in one of the three configurations shown in Figures 4.a-c. For instance, the connectors in Figures 4.a-c can be created as follows. We can create two channels of types `t1` and `t2` by `<a, b> = create(t1)` and `<c, d> = create(t2)`. The connectors in Figures 4.a-c are constructed out of such two channels by performing the operations `join(b, c)`, `join(b, d)`, and `join(a, c)`, respectively. Observe that the channel ends `a`, `b`, `c`, and `d` used in these `join` operations (or any other operation that expects a node rather than a channel end) are merely short-hand for the nodes $\widehat{\mathtt{a}}$, $\widehat{\mathtt{b}}$, $\widehat{\mathtt{c}}$, and $\widehat{\mathtt{d}}$, respectively.

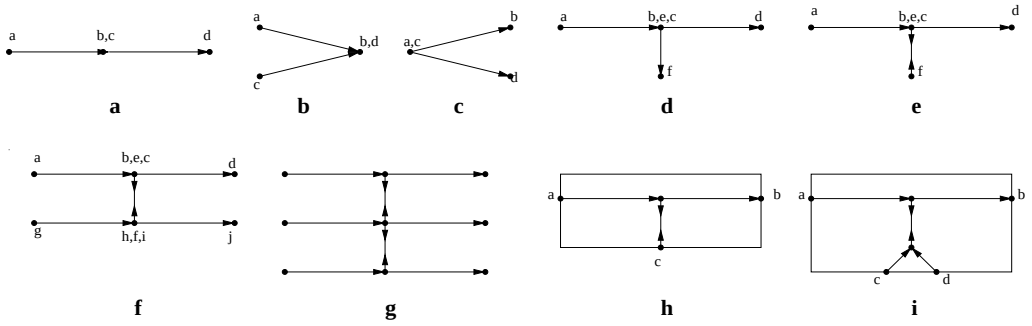

Fig. 4. Examples of channel composition and connectors

10.1. *Flow-through Connectors*

In this section we show how the informal semantics of Reo supports our intuitive expectation of the behavior of the connector in Figure 4.a: that it simply allows data items to flow through the junction node, from the channel `ab` to the channel `cd`. Let

$$N = \widehat{\mathtt{b}} = \widehat{\mathtt{c}}. \tag{14}$$

Because N is not a source node and $Snk(N) = \{\mathtt{b}\}$, from equation 2 we have

$$\mathtt{offers}(N, *) = \mathtt{offers}(\mathtt{b}, *). \tag{15}$$

Similarly, because N is not a sink node and $Src(N) = \{\mathtt{c}\}$, equation 1 gives

$$\mathtt{accepts}(N, d) = \mathtt{accepts}(\mathtt{c}, d). \tag{16}$$

Equations 15 and 16 together simplify equation 3 into

$$\tau(N) = \{\langle y, d\rangle \mid \langle y, d\rangle \in \mathtt{offers}(\mathtt{b}, *) \wedge \mathtt{accepts}(\mathtt{c}, d)\}. \tag{17}$$

Consider the semantics of the mixed node N as presented in Table 8. The behavior of

the channels defined in Section 7 shows that $\texttt{offers}(\texttt{b}, *)$ can contain only pairs of the form $\langle \texttt{b}, z \rangle$. Thus, $\langle y, d \rangle$ on line 3 can match only if $y = \texttt{b}$.

By equation 17, the `_take` operation on line 4 in Table 8 removes every data item d for which $\langle \texttt{b}, d \rangle \in \texttt{offers}(\texttt{b}, *)$ and $\texttt{accepts}(\texttt{c}, d)$ holds. This removes every data item d from the channel `ab` for which $\texttt{accepts}(\texttt{c}, d)$ is true. Because $Src(\widehat{c}) = \{\texttt{c}\}$, the only value that the variable x can assume on line 5 is $x = \texttt{c}$, which means the `_write` operation on line 6 executes only once for this value of x. Because $\texttt{accepts}(\texttt{c}, d)$ is true, the $\texttt{\_write}(\infty, \texttt{c}, d)$ operation on line 6 succeeds in finite time to write the data item d into the channel `cd`.

10.2. *Merger*

The configuration of channels in Figure 4.b allows `write` operations on `a` and `c`, and `read` or `take` operations on `b` and `d`; the channel ends `b` and `d` can be used interchangeably, because they both stand for their common node. A `read` or `take` from this common node delivers a value out of `ab` or `cd`, chosen nondeterministically, if both are non-empty. Thus, assuming the channels are not lossy, this connector produces through the common node of `b` and `d`, a nondeterministic merge of the values that arrive on `a` and `b`.

10.3. *Replicator*

The configuration of channels in Figure 4.c allows `write` operations on `a` and `c`, wherein the two channel ends are interchangeable, and `read` or `take` operations on `b` and `d`. A `write` on (the common node of) `a` (and `c`) succeeds only if both channels are capable of consuming a copy of the written data (see the definition of `write` in Table 3). If they are both of type `FIFO`, of course, all writes succeed. However, if even one is not prepared to consume the data, the write suspends.

10.4. *Take-Cue Regulator*

The significance of the "replication on write" property in Reo can be seen in the composition of the three channels `ab`, `cd`, and `ef` in the configuration of Figure 4.d. Assume `ab` and `cd` are of type `FIFO` and `ef` is of type `Sync`. The configuration in Figure 4.d, then, shows one of the most basic forms of exogenous coordination: the number of data items that flow from `ab` to `cd` is the same as the number of `take` operations that succeeds on `f`. Compared to the configuration in Figure 4.a, what we have in Figure 4.d is a connector where an entity can count and regulate the flow of data between the two channels `ab` and `cd` by the number of `take` operations it performs on `f`. The entity that regulates and/or counts the number of data items through `f` need not know anything about the entities that write to `a` and/or `take` from `d`, and the latter two entities need not know anything about the fact that they are communicating with each other, or the fact that the volume of their communication is regulated and/or measured.

10.5. *Write-Cue Regulator*

The composition of channels in Figure 4.e is identical to the one in Figure 4.d, except that now `ef` is of type `SyncDrain`. The functionality of this configuration of channels is identical to that of the one in Figure 4.d, except that now `write` operations on `f` regulate the flow, instead of `take`s.

10.6. *Barrier Synchronizers*

We can use this fact to construct a barrier synchronization connector, as in Figure 4.f. Here, the `SyncDrain` channel `ef` ensures that a data item passes from `ab` to `cd` only simultaneously with the passing of a data item from `gh` to `ij` (and vice versa). If the four channels `ab`, `cd`, `gh`, and `ij` are all of type `Sync`, our connector directly synchronizes `write`/`take` pairs on the pairs of channels `a` and `d`, and `g` and `j`. This simple barrier synchronization connector can be trivially extended to any number of pairs, as shown in Figure 4.g.

10.7. *Encapsulation and Abstraction*

Figure 4.h shows the same configuration as in Figure 4.e. The enclosing box in Figure 4.h introduces our graphical notation for presenting the encapsulation abstraction effect of the `hide` operation in Reo. The box conveys that a `hide` operation has been performed on all nodes inside the box (in this case, just the one that corresponds to the common node of the channel ends `b`, `c`, and `e` in Figure 4.e). As such, the topology inside the box is immutable, and can be abstracted away: the whole box can be used as a "connector component" that provides only the connection points on its boundary. In this case, assuming that the channels connected to `a` and `b` are of type `Sync`, the function of the connector can be described as "every `write` to `c` enables the transfer of a data item from `a` to `b`.

Through parameterization, the configuration and the functionality of such connector components can be adjusted to fit the occasion. For instance, Figure 4.i shows a variant of the connector in Figure 4.h, where a `write` to either `c` or `d` enables the transfer of a data item from `a` to `b`. The Reo code that instantiates a generic connector of this type is shown in Table 9. The parameter `n` specifies the number of desired regulator points. The return value of a call to this function is a triple that contains the identities of the connector's primary input and output nodes, followed by a sequence of the identifiers for its `n` regulator nodes. A `WCRegulator(1)` call produces (a slightly modified version of) the connector shown in Figure 4.h. A `WCRegulator(2)` call produces the connector shown in Figure 4.i.

10.8. *Ordering*

The connector in Figure 5.a consists of three channels: `ab`, `ac`, and `bc`. The channels `ab` and `ac` are `SyncDrain` and `Sync`, respectively. The channel `bc` is of type `FIFO1`. Let us consider the behavior of this connector, assuming a number of eager producers

```
1   WCRegulator(n)
2     ⟨a, x1⟩ = create(Sync)
3     ⟨x2, b⟩ = create(Sync)
4     ⟨x, y⟩ = create(SyncDrain)
5     connect(x1)
6     connect(x2)
7     join(x, x1)
8     join(x1, x2)
9     hide(x)
10    c = ⟨⟩
11    for i = 1 to n do
12      ⟨u, w⟩ = create(Sync)
13      c = c ∘ ⟨u⟩
14      connect(w)
15      join(y, w)
16    done
17    hide(y)
18    return ⟨a, b, c⟩
```

Table 9. *Reo code for a generic Write-Cue Regulator connector*

and consumers are to perform `write` and `take` operations on the three nodes in this connector. Observe that it is irrelevant whether the producers and consumers in question are component instances that perform `write` and `take` operations, or alternatively, other channels with available data items and available channel capacities. However, to simplify our presentation, we assume the nodes of our connector are connected to appropriate component instances that are prepared to perform suitable `write` and `take` operations on them.

The nodes `a` and `b` can be used (successfully) in `write` operations only; and the node `c` can be used (successfully) only in `take` operations. A `write` on either `a` or `b` will remain pending at least until there is a `write` on both of these nodes; it is only then that both of these operations can succeed simultaneously (because of the `SyncDrain` between `a` and `b`). For a `write` on `a` to succeed, there must be a matching `take` pending on `c`, at which time the value written to `a` is transferred and consumed by the `take` on `c`. Simultaneously, the value written to `b` is transferred into the `FIFO1` channel `bc` (which is initially empty, and thus can consume and hold one data item). As long as this data item remains in `bc`, no other `write` operations can succeed on `a` or `b`; the only possible transition is for another `take` on `c` to consume the contents of the `bc` channel. Once this happens, we return to the initial state and the cycle can repeat itself.

The behavior of this connector can be seen as imposing an order on the flow of the data items written to `a` and `b`, through `c`: the data items obtained by successive `take` operations on `c` consist of the first data item written to `a`, followed by the first data item written to `b`, followed by the second data item written to `a`, followed by the second data item written to `b`, etc. We can summarize the behavior of our connector as $c = (ab)*$, meaning the sequence of values that appear through `c` consist of zero or more repetitions of the pairs of values written to `a` and `b`, in that order. Observe that the a and the b in the expression $(ab)*$ do not represent specific values; rather, they refer to

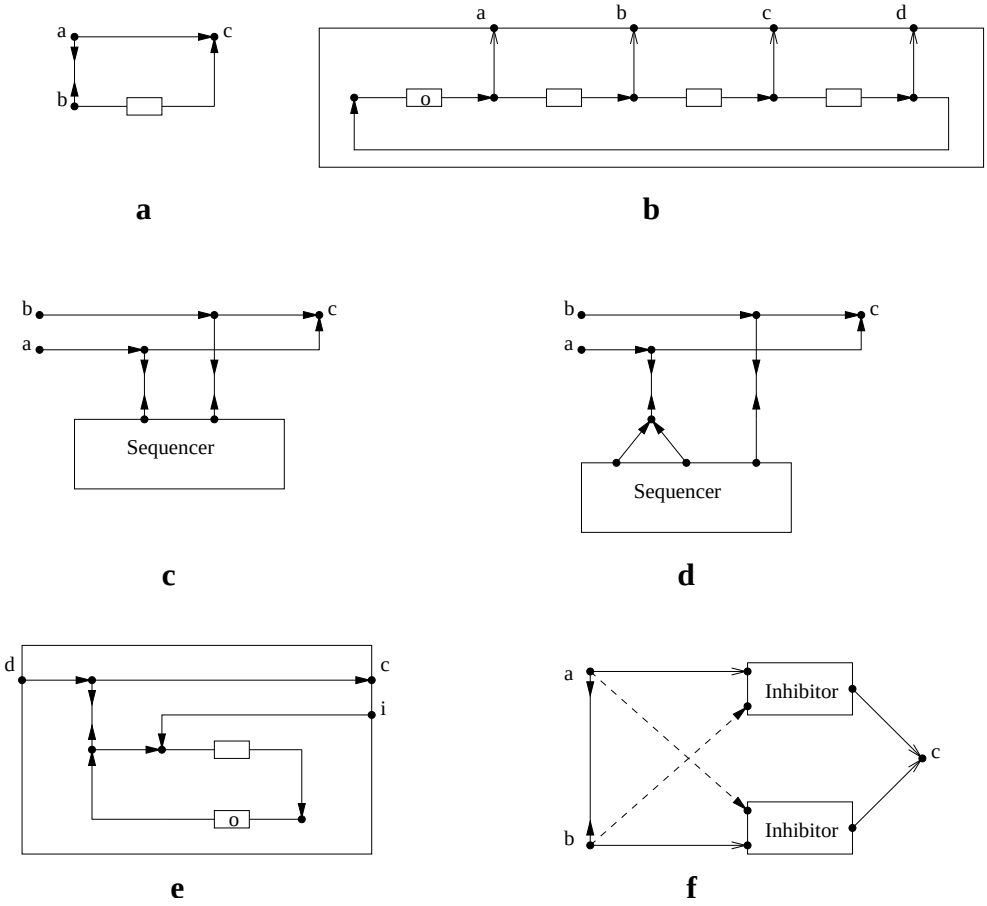

Fig. 5. Connectors for more complex coordination

the `write` operations performed on their respective nodes, irrespective of the actual data items that they write. In other words, we may consider the expression $(ab)*$ not as a regular expression over values, but rather as a meta-level regular expression over the I/O operations that produce (isomorphic) sequences or streams of values on their respective nodes.

11. Expressiveness

The producers and consumers connected to the nodes `a`, `b`, and `c` of the connector in Figure 5.a are completely unaware of the fact that this connector coordinates them through their innocent `take` and `write` operations to impose a specific ordering on them. This interesting coordination protocol emerges due to the composition of the specific channels that comprise this connector in Reo. It is natural at this point to wonder about the expressiveness of the composition paradigm of Reo, i.e., given a (small) set of primitive

channel types, what coordination patterns can be implemented in Reo by composition of such channel types?

In this section we demonstrate, by examples, that Reo connectors composed out of only five simple basic channel types can (exogenously) impose coordination patterns that can be expressed as regular expressions over I/O operations on their nodes. These five channel types consist of `Sync`, `SyncDrain`, `LossySync`, `AsyncDrain`, and an asynchronous channel with the bounded capacity of 1 (e.g., `FIFO1`).¶

11.1. *Sequencer*

Consider the connector in Figure 5.b. As before, the enclosing box represents the fact that the details of this connector are abstracted away and it provides only the four nodes `a`, `b`, `c`, and `d` for other entities (connectors and/or component instances) to (in this case) take from. Inside this connector, we have four `Sync` and four `FIFO1` channels connected together. The first (leftmost) `FIFO1` channel is initialized to have a data item in its buffer, as indicated by the presence of the symbol "o" in the box representing its buffer. The actual value of this data item is irrelevant. The `take` operations on the nodes `a`, `b`, `c`, and `d` can succeed only in the strict left to right order. This connector implements a generic sequencing protocol: we can parameterize this connector to have as many nodes as we want, simply by inserting more (or fewer) `Sync` and `FIFO1` channel pairs, as required. What we have here is a generic *sequencer* connector.

Figure 5.c shows a simple example of the utility of our sequencer. The connector in this figure consists of a two-node sequencer, plus a pair of `Sync` channels and a `SyncDrain` channel connecting each of the nodes of the sequencer to the nodes `a` and `c`, and `b` and `c`, respectively. The connector in Figure 5.c is another connector for the coordination pattern expressed as $c = (ab)*$. However, there is a subtle difference between the connectors in Figures 5.a and c: the one in Figure 5.a never allows a `write` to `a` succeed without a matching `write` to `b`, whereas the one in Figure 5.c allows a `write` to `a` succeed (if "its turn has come") regardless of the availability of a value on `b`.

It takes little effort to see that the connector in Figure 5.d corresponds to the meta-regular expression $c = (aab)*$. Figures 5.c and d show how easily we can construct connectors that correspond to the Kleen-closure of any "meta-word" using a sequencer of the appropriate size. To have the expressive power of regular expressions, we need the "or" as well.

11.2. *Inhibitor*

The connector in Figure 5.e is an *inhibitor*: values written to `d` flow freely through to `c`, until some value is written to `i`, after which the flow stops for good.

¶ Observe that with capacity of 1, the (FIFO or any other) ordering becomes irrelevant. Our specific choice of `FIFO1` here is merely due to the fact that it is the only capacity 1 channel mentioned in this paper.

11.3. *Or Selector*

Our "or" selector can now be constructed out of two inhibitors and two `LossySync` channels, plus some other connector for nondeterministic choice. The connector in Figure 5.f is a particular instance of such an "or" connector. The channel connecting the nodes `a` and `b` in this connector is an `AsyncDrain`. It implements a nondeterministic choice between `a` and `b` if both have a value to offer, and otherwise it selects whichever one arrives first. Each of the nodes `a` and `b` is connected to the inhibitor node of the inhibitor connector that regulates the flow of the values from the other node to `c`. Thus, if a value arrives on `a` before any value arrives on `b`, this connector blocks the flow from `b` to `c` for good and we have $c = a*$. Symmetrically, we have $c = b*$, and we can thus write, in general, $c = a * | b*$.

Observe that the simultaneity-preventing semantics of `AsyncDrain` excludes the possibility of both inhibitors blocking, even if the two initial data items arrive simultaneously at a and b.

12. Formal Semantics

The informal description of the operational semantics of Reo presented in this paper hints at its implementation on a distributed platform. It is, of course, possible to formalize this operational semantics, e.g., in terms of transition systems. An interesting alternative to such a formal semantics, is Rutten's work on a coalgebraic semantics for Reo (Arbab and Rutten, 2002). This work currently covers the core of Reo and we present an overview of its essential features in this section to give a flavor of the types of reasoning that is possible in Reo's coinductive calculus of connectors. In Rutten's model, Reo connectors are relations on *timed data streams*, which consist of twin pairs of separate data and time streams. Coinduction is the main reasoning principle used to prove properties such as connector equivalence.

A *stream* (over A) is an infinite sequence of elements of some set A. Streams over sets of (uninterpreted) data items are called *data streams* and are typically denoted as α, β, γ, etc. Zero-based indices are used to denote the individual elements of a stream, e.g., $\alpha(0), \alpha(1), \alpha(2), ...$ denote the first, second, third, etc., elements of the stream α. Following the conventions of stream calculus (Rutten, 2001), the well-known operations of head and tail on streams are called *initial value* and *derivative*: the initial value of a stream α (i.e., its head) is $\alpha(0)$, and its (first) derivative (i.e., its tail) is denoted as α'. Relational operators on streams apply pairwise to their respective elements, e.g., $\alpha \geq \beta$ means $\alpha(0) \geq \beta(0), \alpha(1) \geq \beta(1), \alpha(2) \geq \beta(2), ...$ *Time streams* are constrained streams over (positive) real numbers, representing moments in time, and are typically denoted as a, b, c, etc. To qualify as a time stream, a stream of real numbers must be strictly increasing, i.e., if a is a time stream, then the constraint $a < a'$ holds.

A *timed data stream* is a pair, $\langle \alpha, a \rangle$, of time ($a$) and data ($\alpha$) streams with the interpretation that $\forall i \geq 0$, the data item $\alpha(i)$ appears at its corresponding time moment $a(i)$. Timed data streams are used to model the flows of data through channel ends. A channel itself is just a (binary) relation between the two timed data streams associated

with its two ends. A more complex connector is simply an n-ary relation among n timed data streams, each representing the flow of data through one of the (non-hidden) n nodes of the connector.

The simplest channel, `Sync`, is formally defined as the relation:

$$\langle \alpha, a \rangle \texttt{ Sync } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b \tag{18}$$

The equation $\alpha = \beta$ states that every data item that goes into a `Sync` channel comes out in the exact same order. The equation $a = b$ states that the arrival and the departure times of each data item are the same: there is no buffer in the channel for a data item to linger on for any length of time.

A `FIFO` channel is defined as the relation:

$$\langle \alpha, a \rangle \texttt{ FIFO } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b \tag{19}$$

As in a synchronous channel, every data item that goes in, comes out of a `FIFO` channel in exactly the same order ($\alpha = \beta$). However, the departure time of each data item is necessarily *after* its arrival time ($a < b$): every data item must necessarily spend some non-zero length of time in the buffer of a `FIFO` channel.

A `FIFO1` channel is very similar to a `FIFO`:

$$\langle \alpha, a \rangle \texttt{ FIFO1 } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b < a' \tag{20}$$

Not only the departure time of every data item, $\alpha(i) = \beta(i)$, is necessarily after its arrival time ($a(i) < b(i)$), but since the channel can contain no more than 1 element, the arrival time $a(i+1)$ of the next data item, $\alpha(i+1)$, must be after the departure time $b(i)$ of its preceding element ($b < a'$).

A `SyncDrain` channel merely relates the timing of the operations on its two ends:

$$\langle \alpha, a \rangle \texttt{ SyncDrain } \langle \beta, b \rangle \equiv a = b \tag{21}$$

The replication that takes place at Reo nodes can be defined in terms of the ternary relation R:

$$R(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \alpha = \beta = \gamma \wedge a = b = c \tag{22}$$

The semicolon delimiter separates "input" and "output" arguments of the relation. The relation R represents the replication of the single "input" timed data stream $\langle \alpha, a \rangle$ into two "output" timed data streams $\langle \beta, b \rangle$ and $\langle \gamma, c \rangle$.

The nondeterministic merge that happens at Reo nodes is defined in terms of the ternary relation M:

$$\begin{aligned} &M(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \\ &\quad a(0) \neq b(0) \wedge \begin{cases} \alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge M(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\ \beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge M(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{otherwise} \end{cases} \end{aligned} \tag{23}$$

The notion of "dense time" represented by real numbers, is more abstract than "discrete time" represented by natural numbers (H. Barringer et al., 1986). In our formal model of Reo, time is strictly local and as such the actual numeric values in time streams do not matter; only their relations are significant. The equality of time moments does not

necessarily imply *simultaneity*, but merely denotes *atomicity*. This means that if for some $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$, we have $a(i) = b(j)$, then $\alpha(i)$ and $\beta(j)$ must appear *atomically*, but one may follow the other; they may actually appear in any order, *as well as* simultaneously, so long as their appearance is not interleaved with the appearance of an unrelated data item. This relaxed interpretation of time enables us to break strict simultaneity, whenever necessary, by shifting the numeric values in selected time streams, as long as atomicity is preserved. For instance, the $a(0) \neq b(0)$ required by our merger, above, can easily be satisfied by nondeterministically shifting one of the two time streams a and b.

Such a simple set of concepts is sufficient to formally derive the properties of the non-trivial connectors presented in this paper (Arbab and Rutten, 2002). For instance, the properties of the regulators of Figures 4.d and e, the barrier synchronizer connector of Figure 4.f, the ordering imposed by the connector in Figure 5.a, the sequencing property of the connector in Figure 5.b, etc., all can be formally derived in this coalgebraic model. Furthermore, this formalism enables us to prove interesting results such as "the pipeline composition of k individual `FIFO1` channels is equivalent to a single `FIFO`k channel," etc. The reader in encouraged to see (Arbab and Rutten, 2002) for details.

13. Conclusion

Reo is an exogenous coordination model wherein complex coordinators, called connectors, are constructed by composing simpler ones. The simplest connectors correspond to a set of channels supplied to Reo. As long as these channels comply with a non-restrictive set of requirements defined by Reo, the semantics of Reo operations, specifically its composition, is independent of the specific behavior of channels. These requirements define the generic aspects of the behavior of channels that Reo cares about, ignoring the details of their specific behavior.

The semantics of composition of connectors in Reo and their resulting coordination protocols can be explained and understood intuitively because of their strong correspondence to a metaphor of physical flow of data through channels. This metaphor naturally lends itself to an intuitive graphical representation of connectors and their composition that strongly resembles (asynchronous) electronic circuit diagrams. Reo connector diagrams can be used as the "glue code" that supports and coordinates inter-component communication in a component based system. As such, drawing Reo connector diagrams constitutes a visual programming paradigm for coordination and component composition.

The topology of connectors in Reo is inherently dynamic and it accommodates mobility. Moreover, Reo supports a very liberal notion of channels. As such, Reo is more general than dataflow models, Kahn-networks, and Petri-nets, all of which can be viewed as specialized channel-based models that incorporate certain specific primitive coordination constructs. Broy's work on timed dataflow channels (Broy and Stefanescu, 2001; Broy and Stolen, 2001) is perhaps closest to Reo. Nevertheless, Reo's more general notion of channels, its inherent dynamic topology, and the fundamental notion of channel/connector composition distinguish it from this model as well.

Connector composition in Reo is very flexible and powerful. Our examples in this paper demonstrate that exogenous coordination protocols that can be expressed as regular

expressions over I/O operations correspond to Reo connectors composed out of a small set of only five primitive channel types.

Our on-going work on Reo in our group includes the formalization of its semantics based on the coalgebraic methodology, which has been developed as a general behavioral theory for dynamical systems. Moreover, we are working on an implementation of Reo to support composition of component based software systems in Java, and the development of logics for reasoning about connectors.

14. Acknowledgment

I am thankful for the discussions and the collaboration of all my colleagues, especially F. Mavaddat, M. Bonsangue, F. de Boer, and J. Guillen Scholten, who have directly or indirectly contributed to the ideas in Reo. I am grateful for the attention and the creative influence of the participants in the ACG seminar series of J. de Bakker at CWI, where various aspects of Reo were presented and discussed in 2001. I am particularly grateful for J. Rutten's keen interest in Reo and his inspiring work on a coalgebraic formal semantics for it. Finally, I immensely appreciate the detailed, careful, and constructive comments of the anonymous reviewers, whose thoughtful suggestions vastly helped to improve this paper.

References

Arbab, F. (1996). The IWIM model for coordination of concurrent activities. In Ciancarini, P. and Hankin, C., editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag.

Arbab, F. (1998). What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, pages 11–22. Available on-line http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief.html.

Arbab, F. (2002). A channel-based coordination model for component composition. Technical Report SEN-R0203, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

Arbab, F., de Boer, F., and Bonsangue, M. (2000a). A coordination language for mobile components. In *Proc. ACM SAC'00*.

Arbab, F., de Boer, F. S., and Bonsangue, M. M. (2000b). A logical interface description language for components. In Porto, A. and Roman, G.-C., editors, *Coordination Languages and Models: Proc. Coordination 2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 249–266. Springer-Verlag.

Arbab, F. and Mavaddat, F. (2002). Coordination through channel composition. In Arbab, F. and Talcott, C., editors, *Coordination Languages and Models: Proc. Coordination 2002*, volume 2315 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag.

Arbab, F. and Rutten, J. J. M. M. (2002). A coinductive calculus of component connectors. Technical Report SEN-R0216, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

Bonsangue, M., Arbab, F., de Bakker, J., Rutten, J., Scutellá, A., and Zavattaro, G. (2000). A transition system semantics for the control-driven coordination language manifold. *Theoretical Computer Science*, 240:3–47.

Broy and Stefanescu (2001). The algebra of stream processing functions. *TCS: Theoretical Computer Science*, 258.

Broy, M. (1995). Equations for describing dynamic nets of communicating systems. In *Proc. 5th COMPASS workshop*, volume 906 of *Lecture Notes in Computer Science*, pages 170–187. Springer-Verlag.

Broy, M. and Stolen, K. (2001). *Specification and development of interactive systems*, volume 62 of *Monographs in Computer Science*. Springer.

de Boer, F. S. and Bonsangue, M. M. (2000). A compositional model for confluent dynamic data-flow networks. In Nielsen, M. and Rovan, B., editors, *Proc. International Symposium of the Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of *Lecture Notes in Computer Science*, pages 212–221. Springer-Verlag.

Grosu, R. and Stoelen, K. (1996). A model for mobile point-to-point data-flow networks without channel sharing. *Lecture Notes in Computer Science*, 1101:504–??

H. Barringer, R. Kuiper, and A. Pnueli (1986). A really abstract current model and its temporal logic. In *Proceedings of Thirteenth Annual ACM Symposium on principles of Programming Languages*, pages 173–183. ACM.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In Rosenfeld, J. L., editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY.

Katis, P., Sabadini, N., and Walters, R. F. C. (2000). A formalization of the IWIM model. In Porto, A. and Roman, G.-C., editors, *Coordination Languages and Models: Proc. Coordination 2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 267–283. Springer-Verlag.

Rutten, J. J. M. M. (2001). Elements of stream calculus (an extensive exercise in coinduction. In Brookes, S. and Mislove, M., editors, *Proc. of 17th Conf. on Mathematical Foundations of Programming Semantics, Aarhus, Denmark, 23–26 May 2001*, volume 45 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam.

Scholten, J. G. (2001). MoCha: A model for distributed Mobile Channels. Master's thesis, Leiden University.