URL: http://www.elsevier.nl/locate/entcs/volume54.html 16 pages

Coordination of Mobile Components

F. Arbab [1]

Software Engineering Department
CWI
Amsterdam, The Netherlands

Abstract

In this paper, we present $P\epsilon\omega$, a paradigm for composition of software components based on the notion of mobile channels. Both components and channels are mobile in $P\epsilon\omega$, in the sense that (1) components can move at any time from one location to another, retaining their existing channel links, and (2) the same channels can be disconnected and reconnected to other components, thus dynamically changing the topology of inter-component communication. The component composition paradigm of $P\epsilon\omega$ is in the style of the IWIM coordination model, and is an extension of our earlier work on a formal-logic-based component interface description language to convey the observable semantics of components. The main focus of attention in $P\epsilon\omega$ is the channels and operations on them, not the processes that operate on them or the components they are connected to. The composition operations in $P\epsilon\omega$ combine various channel types to produce complex dynamic topologies of "connectors" to which processes or components can be attached.

1 Introduction

Many of the issues investigated in the coordination research community in the past decade or so are closely tied to some of the basic problems in Component Based Software Engineering and mobility. Specifically, we believe an IWIM-like coordination model [1,4] can support a powerful channel-based paradigm for composition of software components. Such a paradigm can also easily support the notion of mobility as a general concept that captures both the movement of individual components from one location to the next, leaving the topology of their channel connections intact, as well as the dynamic reconfiguration of the system that changes this topology.

This paper presents $P\epsilon\omega$, a language for composition mobile components using channels as their connectors. Our presentation in this paper is quite informal and is meant to suggest the potential usefulness and the expressive

[1] Email: farhad@cwi.nl

power of a new calculus of channels that we call $P\epsilon\omega$-calculus. The name $P\epsilon\omega$ comes from the Greek word $\rho\epsilon\omega$ which means flow (as of water in streams and channels).

Our work on $P\epsilon\omega$ builds upon and extends our earlier work. In [2] a language for dynamic networks of components is introduced, and in [6] a compositional semantics for its asynchronous subset is given. A formal model for component-based systems is presented in [3], 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 [8]. Generalization of data-flow networks for describing dynamically reconfigurable or mobile networks has also been studied in [5] and [7] for a different notion of observables using the model of stream functions.

2 Basic Concepts in $P\epsilon\omega$

The composition paradigm in $P\epsilon\omega$ consists of three main concepts: locations, components, and channels.

A component is a software implementation that can be executed on a physical or logical device, which we call a location. Components are the basic entities of a system. They can be instantiated at various locations, yielding specific component instances, which interact by means of exchanging values via channels. A component instance may move from one location to another during its life-time.

A channel is a point-to-point medium of peer-to-peer communication. It represents a reliable and directed flow of information from its source to its sink. A component instance may send a value to a channel only if it is connected to its source. Similarly, it may receive a value from a channel only if it is connected to its sink. The identity of the source or the sink of a channel itself can also be communicated via a channel. As such, the connection topology in a system can dynamically change. Initially, we assume that each component instance is connected to a given set of sources and/or sinks of some channels.

Component composition in $P\epsilon\omega$ is accomplished indirectly through channel composition. In addition to the usual read and write operations, $P\epsilon\omega$ includes special composition operators that can be used to construct complex, dynamic dataflow topologies of channels. The constructors of such complex "connectors" in effect coordinate the behavior of the component instances that, perhaps unawarely, read from and write to the available connection points provided by the channel topologies encapsulated in these connectors.

2.1 Patterns

Patterns are used in $P\epsilon\omega$ to regulate channel input/output operations. A pattern is an expression that matches (in the sense of unification in logic programming) an item when it is written to, read from, or simply flows through a channel. 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 (*). 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, string, x>` matches any triplet consisting of the same integer as its first and third element, with a string 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.

A pattern is associated with each channel at its creation time. These patterns restrict the values that can flow through their respective channels. Furthermore, read operations can specify patterns that must match the items they read.

2.2 Channels

A channel is a peer-to-peer medium of communication. Channels are created and destroyed dynamically in $P\epsilon\omega$. A channel has a unique identity and two distinct ends. A channel may follow a synchronous or an asynchronous protocol. An asynchronous channel may have a bounded or an unbounded buffer and may or may not follow a FIFO ordering in the delivery of its contents. Either or both ends of a channel may be connected to component instances, or be dangling. A channel end can simultaneously be known to several component instances, but it cannot be connected to more than a single component instance at any given time.

Channels in $P\epsilon\omega$ are mobile in two senses. (1) If a component instance moves from one location to the next, its attached channel ends also move together with the component instance, preserving the topology of channel connections, without disrupting, affecting, or even the knowledge of the other component instances connected to the opposite ends of these channels. (2) The end of a channel can be disconnected from a component instance, moved and connected to another component instance at the same or another location, without disrupting, affecting, or even the knowledge of the other component instance connected to its opposite end, if any. An implementation of such mobile channels is described in [8] and its Java implementation is currently under way.

There are three types of input/output operations on channels in $P\epsilon\omega$: *read*, *take*, and *write*. The difference between read and take is that a read operation always makes a copy of the available value and does not remove its original from the channel. The take operation, on the other hand, takes the original out of the channel.

2.2.1 Channel Types

$P\epsilon\omega$ assumes the availability of a number of different channel types, each with its own protocol (synchronous or asynchronous) and behavior. One interesting behavior for a channel is when it loses some of the contents it is to carry. Such channels are called *lossy channels*. A channel can be lossy because of the expiration of the time-stamps it requires for all data items it accepts. For example, a (say, FIFO) channel may require an expiration date for every value item written into it, perhaps with a pre-set default. Any value that remains in the channel beyond its expiration date will be automatically deleted by the channel, and can never be read. The second form of a lossy channel is one whose filter pattern does not allow every item written to its source actually enter the channel. While writing such items succeeds normally (i.e., ignoring the filter) as if the channel had actually accepted them, all such items are deleted by the channel automatically. The third form of lossy channels has to do with their bounded capacities: when the bounded capacity of a channel is reached, the arrival of additional data items causes the loss of some data items. A bounded capacity lossy channel can follow a *shift* or an *overflow* regime. Under the shift regime, the arrival of a new data item causes the loss of the oldest data item in channel. Under an overflow regime, the newly arriving data items themselves are lost.

Following is a non-exhaustive list of some interesting channel types in $P\epsilon\omega$.

A `synchronous` channel has a source and a sink and no buffer. Every take or write performed on an end of a synchronous channel hangs until a matching write or take is performed on its opposite end. Once a pair of take and write operations match, and the value item can actually pass through the channel, the value is exchanged and the entities performing these operations continue independently. A read and a write on a synchronous channel also behave the same (as a take and a write), except that because the read operation does not actually take the value item offered by the write operation, the write operation remains pending, while the read succeeds and reads a copy of this same value. If the filter on the channel prevents the value item to flow through the channel, then the write succeeds and the entity performing the write continues independently, while the (read/take) operation on the opposite end of the channel remains pending.

`FIFO` and `bounded` channel types represent the normal unbounded and bounded asynchronous FIFO channels, respectively. The size of the buffer of a `bounded` channel is specified at its creation time. When the buffer of this channel contains the maximum number of items it is allowed to have, a write to its source end suspends until at least one item is taken out of its buffer.

The channel types `bag` and `set` specify asynchronous channels with unbounded buffers that behave as bags (i.e., a multi-sets) and sets, respectively. At most one copy of any value item can exist in the buffer of a `set` at any time. Multiple operations that write the same value item into a `set` all succeed, but the channel will never contain more than one copy of this value item. A `delayset` channel is the same as a `set`, except that an attempt to write a value item that already exists in the buffer of this channel suspends until the existing copy in is taken out.

A `keyedset` channel is an asynchronous channel with an unbounded buffer. Every item written into this channel must be a non-empty tuple. The first element of each tuple is considered as its key. The key value of every tuple in the buffer of a channel of this type must be unique. Writing a tuple whose key value is the same as the key value of an existing tuple, replaces the old tuple in the buffer with the new one.

A `keyedset` channels can be used to construct dynamic records or forms. For instance, such a channel may contain of the tuples `<"FirstName", "Joe">`, `<"LastName", "Blo">`, `<"SocialSecurityNo", "555-12-3456">`, `<"Sex", "Male">`, `<"Age", 46>`, `<"Pets", <"Cat", "Fluffy">, <"Dog", "Spike">, <"Goldfish", "Wanda">>`, `<"Wife", wsnk>`, and `<"Children", c1snk, c2snk, c3snk>`, where `wsnk`, `c1snk`, `c2snk`, and `c3snk`, are references to the sink ends of other keyedset-type channels, containing the records describing Joe Blo's wife and three children.

A `drain` is an asynchronous lossy channel with only two source ends. This channel alternates between its two source ends, every time attempting to consume one item from each. Because this channel has no sink end, everything written to either end of this channel is lost and can never be read (or taken). This channel gives a fair chance to the write operations that may potentially be pending on its two ends, consuming their respective values. However, its alternating behavior guarantees that even when two write operations are pending on its both ends, only one succeeds at a time; this excludes the possibility of simultaneous release of the entities that perform the two write operations. A `syncdrain` (synchronizing drain) behaves the same as a `drain`, except that a write operation on one of its ends suspends until a matching write operation is performed on its opposite end. Only then both write operations succeed simultaneously, and their written values are lost. Both types of drain channels are very useful synchronization tools.

A `spout` is an asynchronous channel with only two sink ends. This channel alternates between its two sink ends, every time making one value item available for taking from each. Because this channel has no source end, nothing can ever be written into it. However, the channel itself acts as an unbounded multi-set of values that can be read or taken out of either of its ends. The values produced out of the ends of a spout, of course, match its respective filter, as well as their corresponding read/take patterns. For instance, reading from a spout sink may produce random integers, perhaps within a range, or repeatedly produce the same constant. Analogous to a drain, the alternating behavior of a spout guarantees fairness but excludes the possibility of simultaneous success at its two ends. A `syncspout` (synchronizing spout) behaves the same as a `spout`, except that a take operation on one of its ends suspends until a matching take operation is performed on its opposite end. Only then both take operations succeed. Both types of spout channels are very useful synchronization tools.

2.3 Components

A component can be instantiated at a location, producing a unique component instance. A component instance in $P\epsilon\omega$ consists of two distinct parts: its interior and its interface. The interior of a component is a black box that conceals its semantics and its static and dynamic structures. The active entities inside the interior of a component instance (e.g., objects, threads, processes, or agents) have a well-defined set of channel operations through which they create and manipulate entities within its own interface, as well as those in that of other component instances.

The only way for (the interiors of) other component instances to communicate with (the interior of) a component instance is through channel connections available within its interface. The interface of a component instance is initialized upon its creation by a parameter substitution mechanism through which its creator makes a number of existing actual channels in the system known to the created component instance. A component instance can dynamically create new channels and communicate their identities with other component instances in the system.

Termination of a component instance releases the memory and other resources allocated to its interior, but its interface remains allocated until it can be ascertained that none of the entities it contains are referenced by entities in the interfaces of other component instances.

2.4 Component Interfaces

The interface of a component instance contains entities that are dynamically created and manipulated by the channel operations performed by the interiors of various component instances in the system. The primary entities contained in a component interface are Channel Ends (CE) and buffer segments that hold the pending contents of asynchronous channels. Channel ends and buffer segments are described elsewhere [8] and we do not elaborate on them here further. For our purposes, it suffices to note that every mobile channel consists of a pair of source- and sink-channel ends, each represented by a CE. A source channel end represents the end of the channel to which data can be written, and a sink channel end is the end of the channel from which data can be read. A channel end may be known to many components, but it can be connected to at most one of them at any given time. Every channel end knows the set of component interfaces that know that channel end.

Channel ends migrate among the interfaces of various components, as a result of the actions initiated by the interiors of their hosts or other component instances. Consequently, were component interiors allowed to refer to channel ends directly, those references would become obsolete upon the migration of their respective channel ends. Furthermore, certain channel operations merge channel ends, which in turn may change what seems to a component to be different channel ends into aliases for the same real channel end. To preserve the integrity of references, component interfaces contain two other types of entities that together support a component-interior's access to a channel end through three levels of indirection.

The only way for the interior of a component to refer to a channel end is through a special data type, called Channel End Variable (CEV), which has two subtypes: `inp` and `outp`. The value of a variable of type `inp` is an indirect reference to a sink channel end. Analogously, the value of a variable of type `outp` is an indirect reference to a source channel end. The interior of a component instance can arbitrarily replicate and use CEV values (i.e., variables of type `inp` and `outp`) within that component instance. Regardless of what happens to the channel ends they ultimately refer to, the indirection mechanism ensures that the CEV values remain meaningful.

There are three ways in which a component instance can become to know a channel end. The first is when the component instance creates a new channel end, e.g., by creating a new channel, and receives its identity. The second is when a component instance reads a value that represents a channel end from another channel end. The third is when the component passes a channel end as an actual parameter to a new component instance.

3 Channel Operations

$P\epsilon\omega$ provides a number of operations on channels and channel ends, most of which are predictable. The operation `create(chantype[, filter])` create a channel with the wildcard (`*`) or the specified `filter` as its filter, and returns the CEV pair that identifies its two ends.

The operation `connect([t,] cev)` connects the specified channel end, `cev`, to the component instance that contains the active entity that performs it. If the channel end is currently connected to another component instance, then only the active entity making the request (not the component instance that it is a part of) suspends and waits in a FIFO queue for the channel end to become disconnected and available for its connection. The optional argument `t` specifies a timeout greater than or equal to zero: if the requested connection is not established before the specified timeout, the request is withdrawn and the suspension of the requesting entity ends. The `disconnect(cev)` operation disconnects the specified channel end from the component instance it is currently connected to.

The operation `wait([t,] cev, conds)` suspends the active entity that performs it, either indefinitely or until the specified timeout, `t`, waiting for the conditions specified in `conds` to become true for the channel end `cev`. The channel end `cev` need not be connected to the component instance for this operation to succeed. The argument `conds` is a boolean combination (using and, or, not, and parenthesis for grouping) of a predefined set of primitive wait conditions that includes `open`, `closed`, `connected`, `disconnected`, `empty`, and `full`.

The operation `read([t,] inp, v[, pat])` suspends the active entity that performs it, either indefinitely or until the specified timeout, `t`, waiting for any value, or one that can match with `pat`, to become available for reading from the channel end `inp` into the variable `v`. The channel end `inp` must be connected to the component instance, and its filter must match with `pat`, if specified, for the read to succeed. Reading a value from a channel does *not* remove that value from the channel. The same value can be read multiple times (even from a synchronous channel). The `take([t,] inp, v[, pat])` operation is the destructive variant of `read`.

The `write([t,] outp, v)` operation suspends the active entity that performs it, either indefinitely or until the specified timeout, `t`, until it succeeds to write the value of the variable `v` to the channel end `outp`. If `v` contains a CEV value, the actual value written to `outp` is a reference to the channel end that `v` refers to. The channel end `outp` must be connected to the component instance for the write to succeed. Writing a value to a synchronous channel unblocks only when it is taken (not just read) from the channel.

4 Connectors

A connector is a set of channels configured in a graph. The vertices in this graph are called *nodes* and its edges are called *paths*. Zero or more channel ends coincide on every node, and there is a path between two nodes x and y if and only if there is a channel whose ends coincide on x and y. All channel ends that coincide on a node are aliases for that node.

A channel itself is a simple example of a connector with two nodes and a path. More complex connectors can be built in $P\epsilon\omega$ using a number of channel composition operators. These operators are best seen as operations that change the topologies of connectors and their accessibility to the components.

In practice, we adorn the undirected edges of the graphs of connectors with direction arrow heads that indicate the flow direction of data items in the channels they represent. The resulting adorned graph is not exactly a directed graph, because of the potential existence of drains and spouts: an edge representing a drain or a spout is always adorned with two opposing arrow heads.

4.1 Nodes

From the viewpoint of a component performing a $P\epsilon\omega$ operation, a node whereupon two or more channel ends coincide is generally indistinguishable from a single channel end. The semantics of most $P\epsilon\omega$ operations defined on channel ends can be extended to nodes in a straight-forward fashion. However, reading from and writing to a node (with more than one coincident channel end), as well as the coincidence of source and sink channel ends at the same node, require special treatment. These configurations have their own special semantics in $P\epsilon\omega$, as described in this section.

A *readable* channel is either (1) a synchronous channel with a write operation pending on its source offering a data item compatible with the channel filter, or (2) an asynchronous channel with a non-empty buffer. A node with one or more readable coincident channels is called a readable node. A read from a node succeeds only if it is readable. When multiple readable channels coincide at a node, every read from that node non-deterministically selects one of them and reads its value.

A *writable* channel is either (1) a synchronous channel with a read operation pending on its sink, or (2) a non-full asynchronous channel. If all channels whose sources coincide at a node are writable, the node is writable. A write to a node succeeds only if it is writable. When the sources of multiple writable channels coincide at a node, every write to that node replicates its value into each and every one of those writable channels.

A node where a mix of source and sink channel ends coincide cannot be or remain connected to any component. This precludes any component's ability to directly read from or write to such a node. Such a mixed node is conceptually equivalent to a simple process that atomically performs a pair of take/write operations whenever the node is both readable and writable.

4.2 Channel Composition

Channel composition in $P\epsilon\omega$ is accomplished through composition of nodes. The `join(cev1, cev2)` operation combines the two nodes identified by the channel ends `cev1` and `cev2` into a single node. All channel ends that were the aliases for `cev1` and `cev2` before this operation become aliases for the new composite node.

The inverse of `join()` is the `split()` operation, which splits a node into two distinct nodes: the same old (i.e., splitting) node plus a fresh new one. The `split()` operation partitions the set of channel ends that coincide on the splitting node before the operation into two subsets. It leaves one subset of channel ends to coincide with the splitting node and the other to coincide with the fresh new node after the split. The partitioning of the set of pre-split coincident channel ends is specified using the concept of a *quoin*. Thus, a split operation is fully specified by a splitting node and its quoin.

The quoin of a split operation is the set of paths forming the exterior structure at the splitting node. All channel ends corresponding to the paths in the quoin of a split "move out" of the splitting node and coincide on the new node after the split. There are three special cases of the general split operation that are of interest in $P\epsilon\omega$: `split(cev1, cev2)`, `split(cev)`, and `split(cev, ALL)`.

The operation `split(cev1, cev2)` specifies `cev1` as the splitting node and uses another node, `cev2`, to identify the quoin of the split operation. In this case, the quoin of the split is the set of all paths with one end on `cev1` and the other on `cev2`.

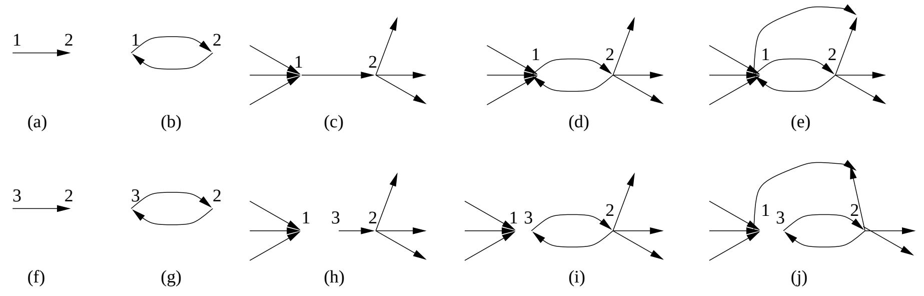

Fig. 1. Examples of join and split operations

Figure 1 shows a few examples of join and split operations. Each of the Figures 1.f through i show the result of performing a `split(1, 2)` on the nodes 1 and 2 in their corresponding Figures 1.a through e. It is the node 1 that splits in each case. The quoin of the splitting operation in each case is the set of all paths with one end coincident on each of the nodes 1 and 2. The effect of the split in Figures 1.a and b is to preserve the topology, while the splitting node 1 becomes private (because node 3 is a newly created node, no other entity can possibly have a reference to it). The splitting of the node 1 in Figures 1.c, d, and e, changes each of their topologies by "moving" the channel ends of the quoin to the new node, 3, producing Figures 1.h, i, and j, respectively. Conversely, `join(1, 3)` turns the connector topologies in Figures 1.h through j into the ones in Figures 1.c through e, respectively, making 3 and 1 aliases for the same joint nodes in each case.

The operation `split(cev)` specifies `cev` as the splitting node and defines the quoin of the split operation as the set of all outgoing paths that coincide on the splitting node. Figure 2 illustrates this form of split (and join). Figure 2.b is obtained from Figure 2.a by joining the (node `x` at the) sink of the left channel with the (node `y` at the) source of the right channel. A `split(x)` (or `split(y)`) operation on the resulting composite node in Figure 2.b produces the topology in Figure 2.a. Joining the nodes `x` and `y` at the source and the sink ends in Figure 2.b produces Figure 2.c. A `split(x)` operation on the resulting composite node in Figure 2.c produces Figure 2.b. Analogously, each pair of constructs in Figures 2.d and e, and Figures 2.f and g are related to each other by join and split operations.

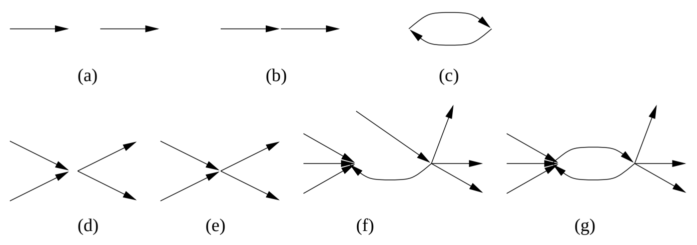

Fig. 2. Examples of join and split with source and sink partitions

The `split(cev, ALL)` operation specifies `cev` as the splitting node and defines the quoin of the split operation as the set of all paths that coincide on the splitting node (i.e., with one end on `cev` and the other on any node in the set of all nodes). The result of this operation is that the splitting node is left with no coincident paths, all of which are moved to the newly created node. All CEV references to the splitting node now become references to a node with no coincident channel ends. All channel ends that coincide on the new node can be referenced only through the single new CEV that is created fresh for this new node, which is known only to the entity that performs the split operation. The splitting node is "privatized" by this operation. In Figures 1.a and b, `split(1, ALL)` has the exact same effect as `split(1, 2)` and produces the same results shown in Figures 1.f and g, respectively.

The `forget(cev)` operation does not change the topology of connectors, but makes the node referenced by `cev` inaccessible through this alias. After this operation, `cev` will no longer be a valid reference to the node it used to refer to.

The combined operation `forget(split(cev, ALL))` is a very important and useful operation. It first privatizes the node referenced by `cev` (i.e., "clones" it, gives the clone node a new name, and moves all paths from `cev` to this clone node). It then forgets the only reference that exists to refer to the resulting private (i.e., clone) node. The net effect is that after this operation, no entity, including the one that performs this operation, can ever refer to the clone node. This ensures that the topology of paths that used to coincide on `cev` cannot be changed anymore.

This combined operation is important enough as an abstraction mechanism to deserve its own special syntax. We define `hide(x)` to be a shorthand for `forget(split(x, ALL))`.

5 Examples

In this section we illustrate the expressive power of $P\epsilon\omega$ with a number of examples.

5.1 Composite Connectors

The channel topologies in Figure 3 show a number of connectors that are often useful as coordination tools for inter-connecting components. A number of channels joined at a common sink (Figure 3.a) is a non-deterministic merger. The sequence of values that come out of their composite sink is a non-deterministic shuffle of the values available through each channel.

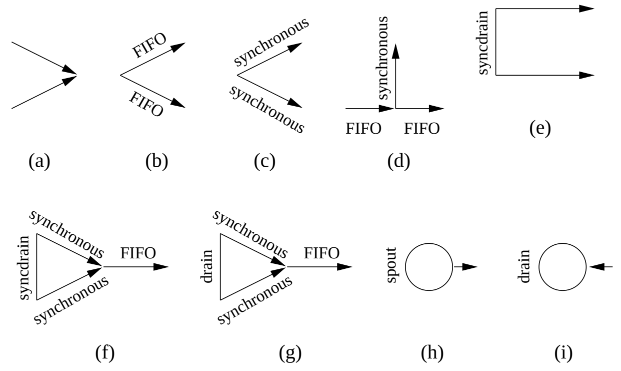

Fig. 3. Channel topologies of some composite connectors

The construct in Figure 3.b shows a replicator. Every item written to the composite source is replicated, with one copy going through each channel.

A number of synchronous channels joined at a composite source produce a readers' barrier synchronizer (Figure 3.c). A value available at their common source can be taken from their individual sinks only when there is a take pending on every sink.

The construction in Figure 3.d shows a flow regulator. The number of value items that pass from the FIFO channel on the left, through the composite channel end, to the FIFO channel on the right is exactly equal to the number of take operations performed on the sink of the synchronous channel.

The construction in Figure 3.e shows a writers' barrier synchronizer. A write operation on either source succeeds only if another pending write simultaneously succeeds on all other source ends.

The topology in Figure 3.f shows a blocking construct. The synchronizing drain channel delays a write operation on either of its ends until there is a write operation pending on both. At this time, these write operations can succeed only if they can both succeed simultaneously. Their simultaneous success, however, is prevented because the ends of the two synchronous channels coincide on the same node: only one data item can be taken out of a node at a time. Consequently, this construct blocks the flow of data items altogether.

The topology in Figure 3.g shows an alternator. Every other item in the FIFO channel comes from the same composite source.

Figure 3.h shows a spout whose sinks are joined together. The composite sink of such a spout is a spring source of value items.

Observe that a synchronizing spout whose sinks are joined together can never produce any values, for the same reason that the construct in Figure 3.f blocks the flow of data. Such an "empty spout" is sometimes useful in the construction or simplification of complex connector topologies.

Figure 3.i shows a drain whose sources are joined together. The composite source of such a (normal or synchronizing) drain is a black hole that consumes every item written to it.

Joining the source and the sink of a FIFO channel produces a repeater device. A number of items can be first written into its resulting composite channel end. Subsequent take operations from this channel end produces an infinite sequence of items that consists of the copies of the items saved in the channel, appearing in the order in which they were written.

5.2 Manifold

The constructs in the coordination language Manifold [1,4] can be described in $P\epsilon\omega$ in a straight-forward manner. A port in Manifold is simply a synchronous channel. Following the Manifold's rules of access, the source ends of input ports and the sink ends of output ports are publicized for access from the outside of their owner processes, while their opposite ends are kept for private. A Manifold stream is a process that administers a $P\epsilon\omega$ FIFO channel. The main function of this administrator process is to force the Manifold's prescribed disconnection at one end of a stream when the connection at its other end breaks. Connection of a stream to a port in Manifold is a join of the respective ends of their corresponding FIFO and synchronous channels in $P\epsilon\omega$. Disconnection in Manifold is a split in $P\epsilon\omega$.

The event-based communication of Manifold can be emulated through special channels in $P\epsilon\omega$. We stipulate a special pair of "event-in" and "event-out" ports for every process through which it receives the event occurrences it is interested in. The (static or dynamic) subscription of a process to an event source is modeled in $P\epsilon\omega$ by connecting the event-out port of the event source to the event-in port of the observer process by a FIFO channel. Raining an event, then, multi-casts the message (i.e., the event occurrence) to all (subscriber) processes currently connected to the event-out port of an event source.

5.3 Tuple Spaces

A Linda-like tuple space can be constructed in $P\epsilon\omega$ using a bag channel type. A component instance that represents a tuple space behaves as follows. It creates a private bag channel that only its own internal entities (presumably different processes) have access to. The component instance then creates a sign-on FIFO channel and publicizes its source end (`ts`) as the identifier for the tuple space it represents. This component instance then becomes the (possibly distributed) server for the tuple space it represents. Any component instance that wishes to perform any Linda-like operation on a specific tuple space, must specify its respective sign-on FIFO channel source end in those operations.

The server component continuously takes individual requests that other components write to its sign-on channel and serves them. Every sign-on request is expected to be just a reference to the source end (`rdsrc`) of an answer FIFO channel. The server may spawn off a new process to serve every request. The reaction of the server (process) to a sign-on request is to create a new query FIFO channel and write the reference to the source end (`wt`) of this query channel into the answer channel (`rdsrc`). This completes the server's part of the sign-on protocol, subsequent to which the server (process) suspends on the sink end of the query channel (i.e., the opposite end of `wt`), waiting for the actual request to arrive.

The following algorithm shows how a Linda-like read operation can be implemented in $P\epsilon\omega$ in our setting. This algorithm takes the identifier of the tuple space, `ts`, a variable, `v`, and a pattern, and returns in `v` a value in the tuple space `ts` that matches with that pattern.

```
Lrd(outp ts, var v, pattern pat)
{
  <rdsrc, rd> = create(FIFO)
  disconnect(rdsrc)
  write(ts, rdsrc)
  take(rd, wt)
  connect(wt)
  write(wt, <"read", pat>)
  take(rd, v>)
  delete(rd, wt)
}
```

This algorithm first creates a new FIFO channel (the answer channel) and writes a reference to its source into the sign-on channel (i.e., `ts`). It then waits to receive a reference to the source end of the query channel (`wt`) through this answer channel (i.e., `rd`). Once it receives the identity of this channel source, it can carry on a private communication with the server (process assigned to its request) through their dedicated pair of query-answer channels. In this case, the `Lrd` algorithm sends a "read" request, together with the specified pattern.

When the server (process) receives this request, it performs a read with the specified pattern on the private bag channel of the server component. The server then writes the result of this read to the answer channel, allowing `Lrd` to read this result through `rd` into the variable `v`.

6 Conclusion

Channels provide peer-to-peer, anonymous communication. This makes channels a good basis for a communication paradigm for component based systems. The topology of channels used in such a system closely represents its architecture. Composition of the variety of channels in $P\epsilon\omega$ produces a powerful set of "connectors" for coordination of the component instances that simply attempt to perform input and output operations on their respective channel ends. Furthermore, channels can easily support dynamic reconfigurability and mobility.

Our current work on $P\epsilon\omega$ is proceeding on two fronts. On the one hand, we are building an implementation platform based on mobile channels to support and experiment with the constructs in $P\epsilon\omega$. On the other hand, we are working on formal models for mobile channels, component based systems using them as their coordination glue, and their semantics, towards a calculus of channels.

7 Acknowledgment

The work presented in this paper has evolved out of my discussions with a number of my colleagues, especially Marcello Bonsangue and Frank de Boer, for whose on-going valuable collaboration on $P\epsilon\omega$ I am grateful. I am also thankful for Juan Guillen Scholten's thesis work on MoCha, which has also helped to clarify some implementation issues.

References

[1] F. Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.

[2] F. Arbab, F.S. de Boer, and M.M. Bonsangue. A coordination language for mobile components. In *Proc. ACM SAC'00*, 2000.

[3] Farhad Arbab, F. S. de Boer, and M. M. Bonsangue. A logical interface description language for components. In Antonio Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models: Proc. Coordination 2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 249–266. Springer-Verlag, September 2000.

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

[5] M. Broy. 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, 1995.

[6] F. S. de Boer and M. M. Bonsangue. A compositional model for confluent dynamic data-flow networks. In M. Nielsen and B. Rovan, 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, August-September 2000.

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

[8] Juan Guillen Scholten. Mocha: A model for distributed Mobile Channels. Master's thesis, Leiden University, May 2001.