Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

A channel-based coordination model for component composition

F. Arbab

REPORT SEN-R0203 FEBRUARY 28, 2002

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

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

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

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

ISSN 1386-369X

A Channel-based Coordination Model for Component Composition

Farhad Arbab
email: farhad@cwi.nl
CWI, P.O. Box 94079, 1090 GB 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. $P\epsilon\omega$ is a channel-based exogenous coordination model wherein complex coordinators, called *connectors* are compositionally built out of simpler ones. The simplest connectors in $P\epsilon\omega$ are a set of *channels* with well-defined behavior supplied by users. $P\epsilon\omega$ 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 $P\epsilon\omega$ is on connectors and their composition only, not on the entities that connect to, communicate, and cooperate through these connectors. Each connector in $P\epsilon\omega$ 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 $P\epsilon\omega$ is a very powerful mechanism for construction of connectors We demonstrate the expressive power of connector composition in $P\epsilon\omega$ 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 $P\epsilon\omega$ out of a small set of only five primitive channel types.

2000 ACM Computing Classification: C.2.4, D.1.3, D.1.m, D.3.2, D.3.3, F.1.2
Keywords and Phrases: Coordination, IWIM, Automata, Fibration.

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. $P\epsilon\omega$ 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.

The characteristics of channel-based models are attractive from the point of view of coordination. Dataflow models, Kahn networks [12], and Petri-nets can be viewed as specialized channel-based models that incorporate certain basic constructs for primitive coordination. IWIM [1, 13] is an example of a more elaborate coordination model based on channels, and Manifold [2, 8] 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 [3]. 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

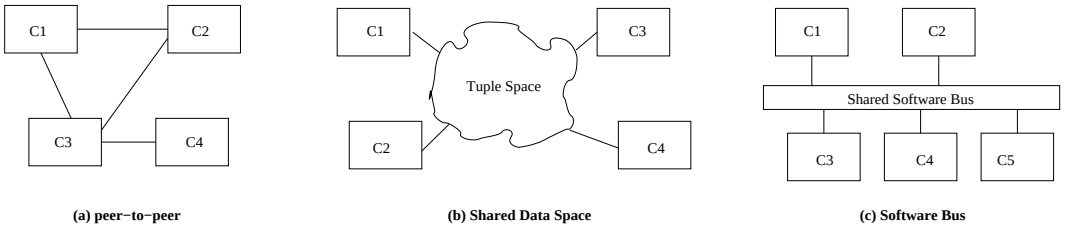


Figure 1: Architectural expressiveness

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 $P\epsilon\omega$, a channel-based model for exogenous coordination introduced in [6, 5]. The name $P\epsilon\omega$ 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, $P\epsilon\omega$ is best transcribed as *Reo*.

Our work on $P\epsilon\omega$ builds upon the IWIM model of coordination and the coordination language Manifold, and extends our earlier work on components. In [4] a language for dynamic networks of components is introduced, and in [10] a compositional semantics for its asynchronous subset is given. A formal model for component-based systems is presented in [7], 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 [14]. Generalization of data-flow networks for describing dynamically reconfigurable or mobile networks has also been studied in [9] and [11] for a different notion of observables using the model of stream functions.

$P\epsilon\omega$ 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. $P\epsilon\omega$ 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 $P\epsilon\omega$ connectors.

The rest of this paper is organized as follows. The basic concepts of components, connectors, channels, etc. are introduced in Section 2. What $P\epsilon\omega$ 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 $P\epsilon\omega$ to define its higher-level operations on connectors. Connectors and channel composition are discussed in Section 4. Section 5 summarizes the set of operations that the (instances of) components can perform. Patterns and channel types are described in Sections 6 and 7, respectively. Sections 8, 9, and 10 provide an insight into the operational semantics of $P\epsilon\omega$ with hints of its actual implementation. Section 11 contains a number of examples of simple connectors constructed out of channels. In Section 12 the expressiveness of the compositional paradigm of $P\epsilon\omega$ 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. Section 13 contains a number of examples of non-trivial connectors constructed out of simpler connectors. In Section 14 we describe how some other coordination models can be emulated in $P\epsilon\omega$. Finally, a summary of our conclusions and future work is presented in Section 15.

2 Basic Concepts

$P\epsilon\omega$ 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 $P\epsilon\omega$. From the point of view of $P\epsilon\omega$, 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 $P\epsilon\omega$ 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.

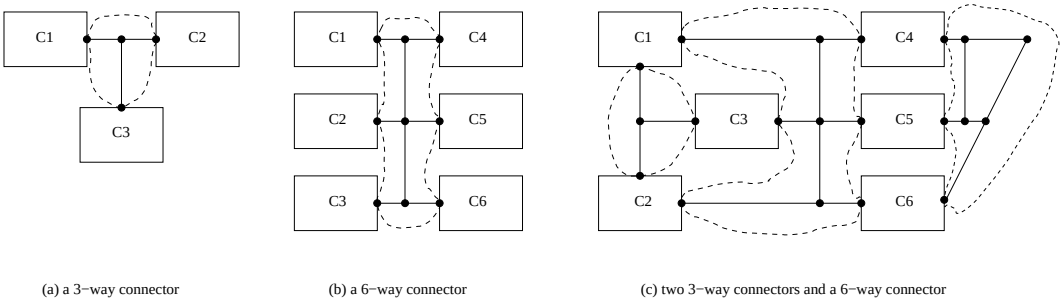


Figure 2: Components and connectors

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 inside a component instance, and the mechanisms used for this communication, are of no interest. Likewise, $P\epsilon\omega$ 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 $P\epsilon\omega$ connectors. Indeed, the constituents inside a component instance may themselves be other component instances that are connected by $P\epsilon\omega$ 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 $P\epsilon\omega$ 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 $P\epsilon\omega$ 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 $P\epsilon\omega$. A channel has its own unique identity. Channels are dynamically created in $P\epsilon\omega$ and they are automatically garbage collected; i.e., they are not explicitly destroyed. A **pattern** is permanently associated with every channel at its creation time as its **filter**. The filter of a channel restricts the set of values that can flow through that channel.

A channel itself has no direction, but each channel in $P\epsilon\omega$ 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 $P\epsilon\omega$ operations, which are described in Section 5.

Channels are used in $P\epsilon\omega$ exclusively to transfer data using input/output operations performed on

Operation	Con.	Description
`create(chantype[, filter])`	-	Creates a channel with the wildcard (`*`) or the specified `filter` as its filter, and returns the identifiers of its two channel ends.
`_forget(cev)`	N	changes `cev` such that it no longer refers to the channel end it designates.
`_move(cev, loc)`	Y	moves `cev` to the location `loc`.
`_connect([t,] cev)`	N	Connects the specified channel end, `cev`, to the component instance that contains the active entity that performs this operation.
`_disconnect(cev)`	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

their ends (specifically, observe that channels do *not* support "message passing" with the "method-call" semantics). A subset of $P\epsilon\omega$ 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 $P\epsilon\omega$. 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 $P\epsilon\omega$ is summarized in Table 1. Operations whose names begin with an underscore are to be used internally by $P\epsilon\omega$ 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. Square brackets are used as meta-symbols to indicate optional parameters. The argument `chantype` designates a channel type, e.g., one of the identifiers that appear in Tables 3 and 4.

- The parameter `filter` is a pattern (see Section 6) that will be assigned to the newly created channel as its filter.
- The parameter `loc` identifies a location.

- The parameter `cev` 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.
- 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 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 $P\epsilon\omega$ 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.

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 $P\epsilon\omega$ 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 `cev` 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 $P\epsilon\omega$ operation `forget` described in Section 4.1.1) causes `cev` 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 cev 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 $P\epsilon\omega$. 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.

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, $P\epsilon\omega$ guarantees the soundness and completeness properties that are shown to be required for compositionality [7].

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, not, and parentheses for grouping) of a set of predefined primitive conditions on channel ends such as _connected(cev), _disconnected(cev), _empty(cev), _full(cev), _contains(cev, pat), etc. 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.

For completeness, $P\epsilon\omega$ requires the negation of every channel condition _*xxx* to also be defined as _not*xxx*. Thus, for the conditions mentioned above, the following negative conditions must also be defined for each channel type: _notconnected(cev), _notdisconnected(cev), _notempty(cev), _notfull(cev), _notcontains(cev, pat), etc. While _notconnected(cev) and _notdisconnected(cev) will always be semantically the same as _disconnected(cev) and _connected(cev), respectively, the relationship between _notempty(cev) and _full(cev), for instance, depends on the specific semantics of different channel types.

The _wait operation applies De Morgan's law on its condition expression to push boolean not 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 either (1) does not match with the filter of the channel to which the source `outp` belongs, or (2) it matches the channel filter and is consumed by the channel.

4 Connectors

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

- Every channel end coincides on exactly one node.
- Zero or more channel ends coincide on every 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 the function $Node(x)$ to designate the unique node on which the channel end x coincides. For a node N, we define

$$Src(N) = \{x \mid x \mapsto N \wedge x$$

is a source channel end$\}$ to be the set of source channel ends that coincide on N. Analogously,

$$Snk(N) = \{x \mid x \mapsto N \wedge x$$

is a sink channel end$\}$ is the set of sink channel ends that coincide on N.

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 $Src(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$.

Observe that 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* $Node(x_c)$ to the edge that represents c. The local direction of the edge representing a channel c in the neighborhood of the node of its sink x_c is presented as an arrow emanating from $Node(x_c)$. Likewise, 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 pointing to $Node(x_c)$.

By definition, every channel represents a (simple) connector. More complex connectors are constructed in $P\epsilon\omega$ out of simpler ones using the `join` operation described in Section 4.1.6.

4.1 Node Operations

The `create` operation in Table 1 inherently deals with channels and channel ends and as such has no counterpart for nodes. Table 2 shows the counterparts of the rest of the operations in Table 1 that work on nodes instead of channel ends.

Operation	Con.	Description
`forget(N)`	N	This operation atomically performs the set of operations `_forget(`x`)`, $\forall x \mapsto$ `N`.
`move(N, loc)`	Y	This operation atomically performs the set of operations `_move(`x`, loc)`, $\forall x \mapsto$ `N`.
`connect([t,] N)`	N	If `N` is not a mixed node, this operation atomically performs the set of operations `_connect([t,], `x`)`, $\forall x \mapsto$ `N`.
`disconnect(N)`	N	This operation atomically performs the set of operations `_disconnect(`x`)`, $\forall x \mapsto$ `N`.
`wait([t,] nconds)`	N	This operation succeeds when the conditions specified in `nconds` become true.
`read([t,] N[, v[, pat]])`	Y	If `N` 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 $x \mapsto$ `N` into the variable `v`.
`take([t,] N[, v[, pat]])`	Y	If `N` 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 $x \mapsto$ `N` and read into the variable `v`.
`write([t,] N, v)`	Y	If `N` 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 $x \mapsto$ `N` atomically.
`join(`N_1`, `N_2`)`	Y	If at least one of the nodes N_1 and N_2 is connected to the component instance, this operation produces a new node that is the result of the destructive merging the two nodes N_1 and N_2 (i.e., N_1 and N_2 no longer exist after the join).
`split(N[, `*quoin*`])`	N	This operation produces a new node N' and splits the set of channel ends that coincide on the node `N` between the two nodes `N` and N' according to the set of edges specified in *quoin*.
`hide(N)`	N	This operation hides the node `N` such that it cannot be used in any other operation.

Table 2: Node operations

The names of the operations in Table 2 do not have underscore prefixes: they are meant to be used by components. The operations in Table 2 are defined only on non-hidden nodes (see Section 4.1.8). They all fail with an appropriate error if any of their node arguments is hidden. As in Table 1, the second column in Table 2 shows whether the connectivity of (all channel ends coincident on) the node argument is a prerequisite for each operation.

4.1.1 Node Forget

A `forget` operation performed by (an active entity inside) a component instance on a node N, atomically performs the set of channel operations `_forget`(x), for all channel ends x that coincide on node N.

4.1.2 Node I/O Operations

A `read`, `take`, or `write` operation performed by (an active entity inside) a component instance becomes and remains *pending* (on that node or its subsequent heirs, as described in Section 4.1.6) until either its time-out expires, or the conditions are right for the execution of its corresponding channel end operation(s). These 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 Section 4.1.4), `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, depend on the generic properties of the channels that coincide on their involved nodes. This is described in Section 9.

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 its node parameter.

4.1.3 Node Move

As defined in Table 2, the `move` operation atomically moves all channel ends that coincide on its node argument to its location argument. 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-$P\epsilon\omega$) 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.

4.1.4 Node Connection

As defined in Table 2, the `connect` and `disconnect` operations atomically connect and disconnect all channel ends that coincide on their node arguments to their respective component instances. Only source and sink nodes (not mixed nodes) can be connected to component instances. Thus, a `connect` fails if its argument node 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.

4.1.5 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 (positive or negative) primitive condition on a channel end `x`, e.g., `_connected(x)`, `_disconnected(x)`, `_empty(x)`, `_full(x)`, etc., $P\epsilon\omega$ defines two corresponding primitive conditions on a node `N`, e.g., `connected(N)` and `connectedAll(N)`, `disconnected(N)` and `disconnectedAll(N)`, `empty(N)` and `emptyAll(N)`, `full(N)` and `fullAll(N)`, etc. A primitive node condition without the `All` suffix is true for a node `N` when its corresponding channel end condition is true for some channel end `x` $\mapsto$ `N`. Analogously, a primitive node condition that ends with the suffix `All`, is true for a node `N` when its corresponding channel end condition is true for all channel ends `x` $\mapsto$ `N`.

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

4.1.6 Node Composition

The composition operation `join(`$\mathtt{N}_1$`,` $\mathtt{N}_2$`)` succeeds only if at least one of the two nodes $\mathtt{N}_1$ and $\mathtt{N}_2$ is connected to the component instance, p, containing the active entity that performs this operation. The effect of `join` is the (destructive) merge of the two nodes $\mathtt{N}_1$ and $\mathtt{N}_2$ into a new node, `N`; i.e., all channel ends that previously coincided on either $\mathtt{N}_1$ or $\mathtt{N}_2$, now coincide on the one and the same node, `N`. If $\mathtt{N}_1$ and $\mathtt{N}_2$ are both connected to p and `N` is not a mixed node, then p remains connected to (all channel ends coincident on) the node `N`; otherwise, p is disconnected from (all of the channel ends that coincide on) the node `N`. In other words, p may lose its previous connection to $\mathtt{N}_1$, $\mathtt{N}_2$, or both, or it may "retain" its previous connection to both of them by remaining connected to their common heir, `N`.

When p loses its previous connection with any of the nodes $\mathtt{N}_1$ and $\mathtt{N}_2$, all `read`, `take`, and `write` operations pending on that node are retracted. Otherwise, all operations pending on $\mathtt{N}_1$ and $\mathtt{N}_2$ become

pending on their common heir, N. Specifically, observe that if, e.g., N_1 is connected to p and N_2 is connected to another component instance, q, then a join(N_1, N_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 N_2. See Section 9 for the semantics of mixed nodes and the semantics of I/O operations on other nodes.

4.1.7 Node Splitting

A split operation performed on a node N produces a new node N′ and divides the set of channel ends that coincide on N between the two nodes N and N′. The newly created node, N′, is not connected to any component instance. The split operation does not require its node argument, N, to be connected to the component instance (that contains the active entity) that performs it. Furthermore, it does not affect the connection of N to any component instance that it may be connected to. Consequently, except when *all* channel ends coincident on N are assigned to N′ after the split, any I/O operation that may be pending on N before the split, remains unaffected and pending on N after the split. In the degenerate case where N is left with no coincident channel ends after the split, any I/O operation pending on this node fails.

Different versions of the split operation, with different signatures, allow different ways of specifying how the coincident channel ends are to be split between the old and the new nodes. 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.

In split(N, S), the parameter S is a set of channel ends and every channel end $x \mapsto$ N such that $x \in S$ is moved to the new node, N′. The operation split(N) moves all sink channel ends $x \mapsto$ N to the new node N′, leaving only source channel ends to coincide on N. The quoin of the split in split(N, M) is defined through the set Q of channels with one end on each of the two nodes N and M: the ends of the channels in Q that coincide on N are moved to N′.

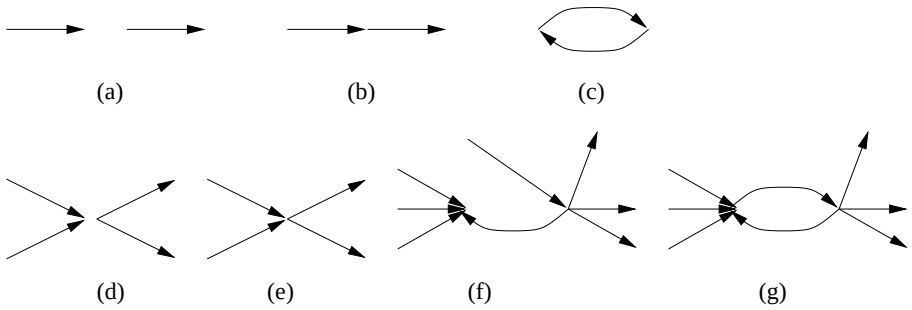


Figure 3: Examples of join and split

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.

4.1.8 Hiding Nodes

The hide(N) operation is an important abstraction mechanism in $P\epsilon\omega$: hiding a node N ensures that N can no longer be used in any other operation (by any active entity in any component instance). This guarantees that the topology of channels coincident on N can no longer be modified by anyone.

5 User-level Operations

User-level operations are the ones that $P\epsilon\omega$ allows the active entities inside component instances to perform. They consist of the set of operations in Table 2, plus the `create` operation in Table 1; i.e., all those operations whose names do not have an underscore prefix. Accordingly, only node (not channel end) conditions can be used by components in wait condition expressions.

For convenience, we extend the operations in Table 2 to also accept channel ends as abbreviations for nodes: a channel end `x` appearing in place of a node `N` in any of the operations in Table 2 stands for the node *Node*(`x`). Analogously, a channel end `x` appearing in place of a node `N` in any of the primitive node conditions in a `wait` operation is treated as an abbreviation for the node *Node*(`x`).

The convenience of using channel ends instead of nodes as arguments of $P\epsilon\omega$ operations has the practical advantage of alleviating the need for components to deal with nodes explicitly as separate entities. The components know and manipulate only channel ends. Channel ends are created by the `create` operation in Table 1, and are passed as arguments to the operations in Table 2, where they actually represent the nodes that they coincide on, rather than specific channel ends. This makes components immune to the dynamic creation and destruction of the nodes whose coincident channel ends they use, while third parties perform `join` and `split` operations on those nodes.

6 Patterns

$P\epsilon\omega$ uses patterns to regulate channel input/output operations. A **pattern** is an expression that matches (in the sense of unification in logic programming) a data item when it is written to, read from, or simply flows through a channel. 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.

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 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.

7 Channel Types

$P\epsilon\omega$ 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 a subset of the data items that it receives, and lose the rest.

Although every channel in $P\epsilon\omega$ 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

Type	Description
`Sync`	has a source and a sink. The pair of I/O operations on its two ends can succeed only simultaneously.
`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`	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 channel filter. Data items are produced in a non-deterministic 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 that match the filter of the channel. 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 3: Examples of synchronous channel types

a channel may depend on such parameters as its filter, its synchronizing properties, the number of its source and sink ends, the size of its buffer, its ordering scheme, its loss policy, etc.

While $P\epsilon\omega$ 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 $P\epsilon\omega$. Tables 3 and 4 show a non-exhaustive set of interesting channel types and their properties. Most of the channel types in these tables are mentioned here as indicative examples only; a few will be used further in this paper as the building blocks for more complex connectors to demonstrate the expressiveness of $P\epsilon\omega$.

7.1 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 both with the filter of c 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 of c; 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 of c as well as the take-pattern p. In the latter case, d is copied into the read-variable v, and the `_take` and the `_write` operations both succeed simultaneously.

7.2 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. A `_write` operation whose value does not match with the channel filter succeeds immediately. All written values are lost.

7.3 Channel Type `SyncSpout`

A `SyncSpout` channel is an unbounded source of data items that match with its channel filter, and can be taken from its opposite ends only simultaneously in some non-deterministic order. While the pair of `_take` operations performed on the opposite ends of a `SyncSpout` 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` each of whose two sink ends `x` and `y` produces an unbounded sequence of integers between 0 and 10 in some non-deterministic 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.

Type	Description
`Ordered`	has a source and a sink, and an unbounded buffer. The source end always accepts all data items that match the filter of the channel. The accepted data items are kept in the internal buffer of the channel. The appropriate operations on the sink end of the channel obtain their matching data items out of the buffer in the same order in which they entered the channel.
`Ordered`n	is the bounded version of `Ordered` with the channel buffer capacity of n data items.
`FIFO`	has a source and a sink, and an unbounded buffer. The source end always accepts all data items that match the filter of the channel. 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.
`Bag`	is similar to `FIFO`, except that its internal buffer is an unbounded bag (multi-set).
`Bag`n	is the bounded version of `Bag` with the channel buffer capacity of n data items.
`Set`	is similar to `Bag`, except that no duplicate data items can exist in the channel buffer, and all such duplicate data items are lost.
`Set`n	is the bounded version of `Set` with the channel buffer capacity of n data items.
`DelaySet`	is similar to `Set`, except that the I/O operation that attempts to insert a duplicate data item into the channel is delayed until its respective data item is taken out of the channel buffer by some other I/O operation.
`DelaySet`n	is the bounded version of `DelaySet` with the channel buffer capacity of n data items.
`KeyedSet`	can accept only tuples of one or more elements as valid data items. The first element in each tuple is considered to be the *key* for that tuple. No two data items with the same key can exist in the channel buffer at the same time. Inserting a new tuple with the same key as that of another tuple already in the channel buffer, replaces the old tuple with the new one.
`KeyedSet`n	is the bounded version of `KeyedSet` 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`	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.
`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 4: Examples of asynchronous channel types

7.4 Channel Types `Ordered` and `Ordered`n

The `Ordered`, and `Ordered`n channel types, where n is an integer greater than zero, represent unbounded asynchronous and bounded asynchronous ordered channels. A `_write` to an `Ordered` channel always succeeds, and a `_write` to an `Ordered`n succeeds onl y if either 1) the written value does not match with the channel filter (in which case it is lost); or otherwise 2) the number of data items in i ts buffer is less than its bounded capacity, n. A `_read` or `_take` from an `ordered` or `ordered`n channel suspends until there is a data item in the channel buffer that matches the `_read` or _take pattern. If multiple matches are possible, the first (i.e., oldest) in the buffer that satisfies the match is selected. The matching data item is then (destructively) obtained and the operation succeeds.

7.5 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 either 1) the written value does not match with the channel filter (in which case it is lost); or otherwise 2) 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.

7.6 Bags and Sets

The channel types `Bag`, `Bag`n, `DelaySet`, and `DelaySet`n, where n is an integer greater than zero, are asynchronous channels with (un)bounded buffer capacities, but unlike `FIFO` and `FIFO`n they do not impose any ordering on the delivery of their buffer contents. A `_read` or `_take` operation on the sink of such a channel succeeds by (non-deterministically) selecting one of the data items in the channel buffer that match the pattern specified in the operation. `Bag` and `Bag`n channels allow multiple copies of the same data item to be present in their buffers at the same time. `DelaySet` and `DelaySet`n channels do not allow duplicates: an attempt to `_write` a duplicate data item into one of these channels suspends until its corresponding data item no longer exists in the channel buffer.

7.7 Channel Types `KeyedSet` and `KeyedSet`n

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.

`KeyedSet` channels are used to construct dynamic records or forms. For instance, a channel of this type may contain 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.

The `KeyedSet`n channel is the bounded version of the `KeyedSet` channel, where the integer $n > 0$ is the maximum capacity of its buffer. Other useful variations of channel types in this family include:

- a *read-only* version of the `KeyedSet` or `KeyedSet`n that ensures that `_take` operations do not actually remove values from its buffer;
- a *delayed* version of the `KeyedSet` or `KeyedSet`n that guarantees that an attempt to `_write` a tuple with the same key as another one already in the channel buffer is delayed until the latter is removed (i.e., by a `_take`); and
- an *immutable* version of the `KeyedSet` that combines the behavior of the *delayed* and the *read-only* versions to make all tuples in the channel buffer immutable (i.e, once a tuple enters the channel, it cannot be removed, nor modified).

7.8 Channel Types `AsyncDrain` and `AsyncSpout`

`AsyncDrain` and `AsyncSpout` are analogous to `SyncDrain` and `SyncSpout`, 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.

7.9 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.

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.

For instance, `Set` and `Set`n channel types are the lossy counterparts of `Bag`, `Bag`n, `DelaySet`, and `DelaySet`n. Like `DelaySet` and `DelaySet`n, they do not allow duplicate data items in their buffers. Unlike `DelaySet` and `DelaySet`n, they do not delay the success of write operations that attempt to insert duplicate data items; instead, they lose the duplicate data items.

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.

An interesting class of lossy channels is annihilator channels. An annihilator channel is an asynchronous channel that partitions a subset of all data items that can enter its buffer into two (not necessarily disjoint) sets of *values* and *anti-values*. Whenever a value and its anti-value happen to be in the buffer of such a channel, they immediately *annihilate* each other and are both lost. For instance, an annihilator bag channel whose filter allows only numbers in its buffer may interpret negative numbers as anti-values for their positive counterparts. Thus, whenever a number x enters this channel, it either (1) remains in the buffer if it does not already contain a $-x$; or (2) the x and a $-x$ are both lost, otherwise. Observe that in this case, the number 0 may or may not be defined as its own anti-value.

8 Channel Behavior

The channel types described in Section 7 are indicative of the richness and the diversity of the behavior of channels allowed in $P\epsilon\omega$. However, $P\epsilon\omega$ is not directly aware of the behavior of any particular channel. $P\epsilon\omega$ expects every channel type to be able to provide a "reasonable implementation" of the operations in Table 1.

The set of operations in Table 1, thus, describes the **common behavior** of all channels in $P\epsilon\omega$. However, the operations in Table 1 are not sufficient to describe the full behavior of different channel types. As far as $P\epsilon\omega$ is concerned, the **generic behavior** of a channel c, whose source and sink are x_c and y_c, respectively, is defined indirectly through three functions:

- `filter`(c) is the filter of the channel c.
- `offers`(y_c, p) is the multi-set of pairs $\langle y_c, d\rangle$ for each d in the multiset of values that may be assigned to a variable v in a `_take`$(0, y_c, v, p)$.
- `takes`(x_c, d) is true for a data item d if $d \ni$ `filter`(c) and 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 `takes`$(y_c, d) = \mathit{false}$, 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 $P\epsilon\omega$ operations are semantically independent of the specific behavior of channels.

9 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 2. Generally, everything flows in $P\epsilon\omega$ 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 $P\epsilon\omega$. In this section, we informally describe the operational semantics of mixed nodes and the read, take, and write operations on nodes. Our exposition is intended to show the fundamentals of a truly distributed implementation of $P\epsilon\omega$. We ignore certain aspects of timing and all locking issues here to simplify our presentation.

For a data item d and the source x_c of a channel c, we define:

$$\texttt{accepts}(x_c, d) = d \not\ni \texttt{filter}(c) \vee \texttt{takes}(x_c, d) \tag{1}$$

and further extend this definition for a node N as:

$$\texttt{accepts}(N, d) = \begin{cases} \mathit{false} & \text{if } N \text{ is a sink node} \\ \bigwedge_{x \in Src(N)} \texttt{accepts}(x, d) & \text{otherwise} \end{cases} \tag{2}$$

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* $\{\texttt{\_write}(\infty, x, d) \mid x \mapsto Src(N) \wedge \texttt{takes}(x, d)\}$ *atomically succeeds, in which case the* `write` *operation succeeds.*

Observe that a `_write` operation is performed only for those channel ends in Definition 1 for which $\texttt{takes}(x, d)$ is true. By the definition of the `take` predicate, above, this guarantees that every such `_write` operation immediately succeeds.

We use the predicate $\Pi(O)$ to designate whether or not the operation O is pending (on its respective node). The special symbol ϵ represents "no channel end" in the following definition. The multi-set of pairs identifying the values offered by a node N is thus defined as:

$$\texttt{offers}(N, p) = \begin{cases} \biguplus_{d \,:\, \Pi(\texttt{write}(t, N, d)) \wedge d \ni p} \{\langle \epsilon, d \rangle\} & \text{if } N \text{ is a source node} \\ \biguplus_{x \in Snk(N)} \texttt{offers}(x, p) & \text{otherwise} \end{cases} \tag{3}$$

According to this definition, a source node offers only the multi-set of values proposed by the write operations pending on that node, each as the second element of a pair whose first element indicates "no channel end." A mixed node cannot be involved in any write operation in $P\epsilon\omega$. Therefore, the multi-set of the values offered by a mixed or a sink node is 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 non-deterministically (but fairly) selected channel end* $x \mapsto N$ *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. By the definition of the `offers` predicate, above, this guarantees that such a `_take` operation succeeds in finite time.

Semantically, a $\texttt{read}(t, N, v, p)$ operation is analogous to $\texttt{take}(t, N, v, p)$, but its details are beyond the scope of 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

$$\tau(N) = \{\langle y, d \rangle \mid \langle y, d \rangle \in \texttt{offers}(N, *) \wedge \texttt{accepts}(N, d)\}. \tag{4}$$

```
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         if (takes(x, d)) then _write(∞, x, d)
7       done
8     done
9   done
```

Table 5: Semantics of a mixed node

Definition 3 *The semantics of a mixed node N in $P\epsilon\omega$ is defined as the execution of the infinite loop in Table 5 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 $\langle y, d\rangle \in$ `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, the `_write`(∞, x, d) operation on line 6 is performed only if `takes`(x, d) is true. This guarantees that the `_write` 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.

10 Generic Behavior of Channels

It is instructive to consider a few common channel types as examples to see how their generic behavior can be expressed in $P\epsilon\omega$. In this section, we describe the behavior of some of the channels in Tables 3 and 4.

10.1 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 the other 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 4 is described in the rest of this section.

10.1.1 Generic Behavior of `Ordered`

Consider an `Ordered` channel c 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 function *filter*(c), which returns the pattern assigned to c as its

filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \begin{cases} \{\langle y_c, B_i \rangle\} & \text{if there exists a smallest } 1 \leq i \leq k \text{ such that } B_1 \ni p \\ \emptyset & \text{otherwise} \end{cases} \quad (5)$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \quad (6)$$

This states that what the sink end of c offers (for reading or taking) is the empty set if the buffer of c contains no suitable matching element, and a singleton (containing the oldest data element that matches p), otherwise.

10.1.2 Generic Behavior of `Ordered`n

The behavior of a `Ordered`n channel is identical to that of a `Ordered`, except that its bounded capacity may prevent it from accepting values when it is full. Let c be a `Ordered`n channel with $B(c) = \langle B_k, B_{k-1}, ...B_2, B_1 \rangle$ representing its buffer, as in Section 10.1.1. 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 function $\mathit{filter}(c)$, which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \begin{cases} \{\langle y_c, B_i \rangle\} & \text{if there exists a smallest } 1 \leq i \leq k \text{ such that } B_1 \ni p \\ \emptyset & \text{otherwise} \end{cases} \quad (7)$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \wedge |B(c)| < n \quad (8)$$

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

10.1.3 Generic Behavior of `FIFO`

Consider a `FIFO` channel c (as described in Table 4) 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 function $\mathit{filter}(c)$, which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \begin{cases} \{\langle y_c, B_1 \rangle\} & \text{if } B(c) \neq \langle\rangle \vee B_1 \ni p \\ \emptyset & \text{otherwise} \end{cases} \quad (9)$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \quad (10)$$

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.

10.1.4 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 10.1.3. 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 function $\mathit{filter}(c)$, which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \begin{cases} \{\langle y_c, B_1 \rangle\} & \text{if } B(c) \neq \langle\rangle \vee B_1 \ni p \\ \emptyset & \text{otherwise} \end{cases} \quad (11)$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \wedge |B(c)| < n \quad (12)$$

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

10.1.5 Generic Behavior of `Bag`

The behavior of a `Bag` channel c is similar to that of a `FIFO`, except that the multi-set $\texttt{offers}(y_c)$ yields all the elements in its buffer that match with the specified pattern. The generic behavior of c, in this case, is defined by the function *filter*(c), which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid d \in B(c) \wedge d \ni p\} \tag{13}$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \tag{14}$$

10.1.6 Generic Behavior of `Bag`n

The behavior of a `Bag`n channel c is the same as that of a `Bag` channel, except that it refuses to take any more values when its bounded buffer capacity is full. The generic behavior of c, in this case, is defined by the function *filter*(c), which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid d \in B(c) \wedge d \ni p\} \tag{15}$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \wedge |B(c)| < n \tag{16}$$

10.1.7 Generic Behavior of `Set` and `Set`n

A `Set` is the same as a `Bag`: the fact that it loses duplicates is part of its specific behavior which does not affect its generic behavior expressed through `offers` and `takes`. The generic behavior of a `Set` channel c is defined by the function *filter*(c), which returns the pattern assigned to c as its filter upon its creation, and the equations 13 and 14, defined in Section 10.1.5.

Analogously, the behavior of a `Set`n channel is defined by its *filter*(c) and the equations 15 and 16, defined in Section 10.1.6.

10.1.8 Generic Behavior of `DelaySet` and `DelaySet`n

A `DelaySet` channel behaves the same as a `Bag`, except that for its $\texttt{takes}(x_c, d)$ to be true, it must also be the case that d does not exist in its buffer. The generic behavior of a `DelaySet` channel c is defined by the function *filter*(c), which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid d \in B(c) \wedge d \ni p\} \tag{17}$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \wedge d \notin B(c) \tag{18}$$

Analogously, the $\texttt{takes}(x_c, d)$ for a `DelaySet`n is defined as:

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \wedge d \notin B(c) \wedge |B(c)| < n \tag{19}$$

10.1.9 Generic Behavior of `KeyedSet` and `KeyedSet`n

The generic behavior of a `KeyedSet` channel c is defined by the function *filter*(c), which returns the pattern assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid d \in B(c) \wedge d \ni p\} \tag{20}$$

$$\texttt{takes}(x_c, d) = d \ni \mathit{filter}(c) \wedge d = \langle k\rangle \circ \alpha \tag{21}$$

The `takes` predicate ensures that the values that can enter the channel are tuples of the form $\langle k \rangle \circ \alpha$, where k is an arbitrary value called *key*, α is an arbitrary possibly empty tuple, and $\circ$ is the concatenation operator. The specific behavior of a `KeyedSet` channel ensures that there is no more than a single tuple with the same key in the channel buffer at any given time, by replacing an old tuple with a new tuple that has the same key.

The behavior of a `KeyedSet`n is derived from that of a `KeyedSet` by imposing the constraint of its bounded capacity in the, by now, obvious manner.

10.1.10 Generic Behavior of `AsyncDrain`

The generic behavior of an `AsyncDrain` channel c with the x_c and y_c as its two source ends is dynamically defined by its specific behavior. Effectively, the two predicates $\texttt{takes}(x_c, p)$ and $\texttt{takes}(y_c, p)$ each alternates between true and false, as the specific behavior of c alternates between a `_take` operation on each of its ends. This guarantees that an `AsyncDrain` channel never consumes data items through its two ends simultaneously. Thus, if its two ends happen to coincide on the same node, it is impossible for them to consume anything at all, because the semantics of flow through nodes requires all consumers on that node to consume their values simultaneously (see Section 9).

10.1.11 Generic Behavior of `AsyncSpout`

The generic behavior of an `AsyncSpout` channel c with the x_c and y_c as its two sink ends is defined dynamically through its specific behavior. Effectively, the two functions $\texttt{offers}(x_c, p)$ and $\texttt{offers}(y_c, p)$ yield singletons or the empty set, as c alternates and performs an immediate `_write` (i.e., with a time-out value of 0) on each of its two ends. This guarantees that an `AsyncSpout` channel never produces data items through its two ends simultaneously.

10.1.12 Generic Behavior of `ShiftFIFO`n and `LossyFIFO`n

The generic behavior of `ShiftFIFO`n and `LossyFIFO`n channels is identical to that of a `FIFO`n 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 $P\epsilon\omega$ operations are concerned, these channel types behave as if they were `FIFO`n channels.

10.2 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 3.

10.2.1 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 function *filter*(c) which returns the patters assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{takes}(x_c, d) = d \ni \texttt{filter}(c) \wedge \Pi(\texttt{\_take}(\infty, y_c, v, p)) \wedge d \ni p \tag{22}$$

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid \langle z, d\rangle \in \texttt{offers}(\mathit{Node}(x_c, p))\}. \tag{23}$$

10.2.2 Generic Behavior of `SyncDrain`

The generic behavior of a `SyncDrain` c whose two source ends are x_c and y_c, is dynamically defined by its specific behavior. Effectively, the two predicates $\texttt{takes}(x_c, p)$ and $\texttt{takes}(y_c, p)$ simultaneously become true, as per its specific behavior, c atomically performs a pair of `_take` operations on its two end.

10.2.3 Behavior or `SyncSpout`

The generic behavior of a `SyncSpout` c whose two sink ends are x_c and y_c, is defined dynamically through its specific behavior. Effectively, the two functions $\texttt{offers}(x_c, p)$ and $\texttt{offers}(y_c, p)$ yield singletons or the empty set, as c atomically performs a pair of `_write` operations on its two ends.

10.2.4 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 function *filter*(c) which returns the patters assigned to c as its filter upon its creation, and the following two functions.

$$\texttt{takes}(x_c, d) = d \ni \texttt{filter}(c) \tag{24}$$

$$\texttt{offers}(y_c, p) = \{\langle y_c, d\rangle \mid \langle z, d\rangle \in \texttt{offers}(\mathit{Node}(x_c, p))\}. \tag{25}$$

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.

11 Channel Composition

The utility of channel composition 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 obtain two channels of types `t1` and `t2`, with filters `f1` and `f2`, by simply creating them: `<a, b> = create(t1, f1)` and `<c, d> = create(t2, f2)`. The connector 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) is merely a short-hand for the nodes *Node*(`a`), *Node*(`b`), *Node*(`c`), and *Node*(`d`), respectively.

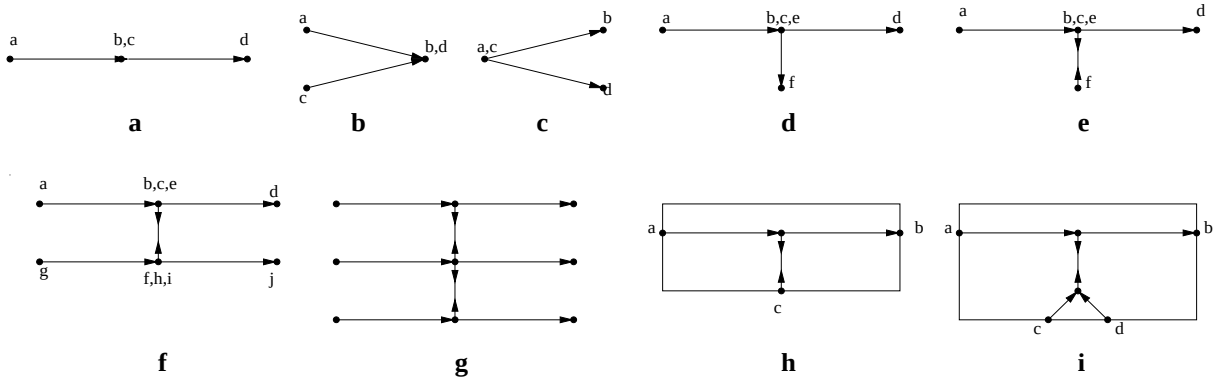


Figure 4: Examples of channel composition and connectors

11.1 Flow-through Connectors

In this section we show how the informal semantics of $P\epsilon\omega$ 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

$$N = \mathit{Node}(\mathtt{b}) = \mathit{Node}(\mathtt{c}). \tag{26}$$

Because N is not a source node and $\mathit{Snk}(N) = \{\mathtt{b}\}$, from equation 3 we have

$$\mathtt{offers}(N, *) = \mathtt{offers}(\mathtt{b}, *). \tag{27}$$

Similarly, because N is not a sink node and $\mathit{Src}(N) = \{\mathtt{c}\}$, equation 2 gives

$$\mathtt{accepts}(N, d) = \mathtt{accepts}(\mathtt{c}, d). \tag{28}$$

Using equation 1 to expand the right-hand-side of equation 28 yields

$$\mathtt{accepts}(N, d) = d \not\ni \mathit{filter}(\mathtt{cd}) \vee \mathtt{takes}(\mathtt{c}, d). \tag{29}$$

Equations 27 and 29 together simplify equation 4 into

$$\tau(N) = \{\langle y, d\rangle \mid \langle y, d\rangle \in \mathtt{offers}(\mathtt{b}, *) \wedge (d \not\ni \mathit{filter}(\mathtt{cd}) \vee \mathtt{takes}(\mathtt{c}, d))\}. \tag{30}$$

Consider the semantics of the mixed node N as presented in Table 5. The behavior of the channels defined in Section 8 shows that $\mathtt{offers}(\mathtt{b}, *)$ can contain only pairs of the form $\langle \mathtt{b}, z\rangle$. Thus, $\langle y, d\rangle$ on line 3 can match only if $y = \mathtt{b}$.

By equation 30, the _take operation on line 4 in Table 5 removes every data item d for which $\langle \mathtt{b}, d\rangle \in \mathtt{offers}(\mathtt{b}, *)$ and $d \not\ni \mathit{filter}(\mathtt{cd}) \vee \mathtt{takes}(\mathtt{c}, d)$ holds. This removes every data item d from the channel ab that either does not match with the filter of the channel cd, or for which $\mathtt{takes}(\mathtt{c}, d)$ is true. Because $\mathit{Src}(N) = \{\mathtt{c}\}$, the only value that the variable x can assume on line 5 is $x = \mathtt{c}$, which means the if-statement on line 6 executes only once for this value of x. If d matches with the filter of cd, then $\mathtt{takes}(\mathtt{c}, d)$ must be true, in which case the $\mathtt{\_write}(\infty, \mathtt{c}, d)$ operation on line 6 succeeds in finite time to write the data item d into the channel cd. If d does not match with the filter of cd, it is simply lost.

11.2 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 non-deterministically, if both are non-empty. Thus, assuming the channels are not lossy, this connector produces through the common node of b and d, a non-deterministic merge of the values that arrive on a and b.

11.3 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. 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.

11.4 Take-Cue Regulator

The significance of the "replication on write" property in $P\epsilon\omega$ 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.

11.5 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 takes.

```
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 6: $P\epsilon\omega$ code for a generic Write-Cue Regulator connector

11.6 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.

11.7 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 $P\epsilon\omega$. 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 $P\epsilon\omega$ code that instantiates a generic connector of this type is shown in Table 6. 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.

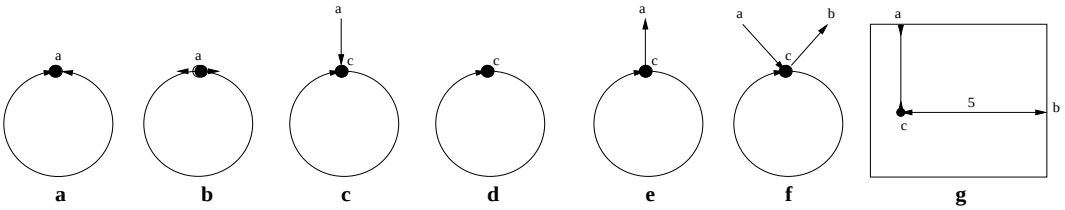


Figure 5: More examples of channel composition and connectors

11.8 Special Spouts

Figure 5.a shows a spout whose two sinks are joined on the same node. A `SyncSpout` channel in this configuration behaves in a peculiar way: each of its sink ends can release a value only if the other end also does so simultaneously. However, because the two sinks coincide on the same node, through which no more than one value can pass at any time, this connector acts as a "dry well" never producing any values.

In contrast, an `AsyncSpout` channel in this same configuration never blocks, because by definition, the channel never produces two values through its two ends simultaneously. Every opportunity to transfer a value from the node succeeds to obtain a value from either one or the other sink end of the channel.

11.9 Special Drains

Figure 5.b shows a drain whose two source ends are joined on the same node. An `AsyncDrain` channel in this configuration behaves in a peculiar way: each of its source ends can consume a value only if the other end does not do so simultaneously. However, because the two sources coincide on the same node, the replication semantics of "write" implies that a transfer to each source end can succeed only if they both consume at the same time. Thus, this connector acts as a blocking device: it never consumes any values.

In contrast, a `SyncDrain` channel in this same configuration never blocks, because every value written (or transferred into) this node will be duplicated and both copies can always be consumed simultaneously, by the definition of the behavior of the `SyncDrain`.

11.10 Repeater

Figure 5.c shows a `FIFO` channel whose ends are joined together with the sink end of another, say `Sync`, channel on a common node, `c`. A write to `a` succeeds immediately, because the value can directly be transferred to `c` and consumed by the source end of the `FIFO` channel. Once a value exists in the `FIFO` channel, it will indefinitely cycle through the node `c`: this node will always find a value is available for taking through a coincident sink node (that of the `FIFO`) and will always be able to write this value to all of its coincident source ends (i.e., that of the `FIFO`). Although this configuration itself is static, once a value is written to this connector, an endless cycle of activity ensues. This cycling continues undisturbed

even if the Sync channel is split away from node c, yielding the configuration in Figure 5.d. However, joining the source end of a Sync channel on to node c, as in Figure 5.e, immediately stops the cycling activity: the value can pass through and cycle only once every time a copy of it can also be consumed through node a.

The FIFO channel in the connector in Figure 5.e acts as a repeater that indefinitely produces copies of its contents; a write in Figure 5.c effectively "loads" the repeater; the Sync channel in Figure 5.e regulates the cycling and the replication to take place at exactly those times when values can be taken through node a. In contrast, if the channel ca in Figure 5.e is also a FIFO, then the cycling and replication process will go on indefinitely, independent of the timing of the activity on node a.

Combining the two configurations in Figures 5.c and e, we have the connector in Figure 5.f, which acts as a shuffle repeater: all values in the looped FIFO channel cc are cycled in sequence and replicated through node c; and every write on node a succeeds and inserts its value somewhere within the current sequence of cycling values, which yields the effect of the shuffling of the order of the values written to a.

11.11 Constant Repeater

Figure 5.g shows a SyncDrain channel ac joined with a SyncSpout channel bc, with their common node, c, hidden to preserve their topology. The number 5 on channel bc is its filter: the only value this spout can produce is the number 5. For all practical purposes, the connector in Figure 5.g can be seen as a single synchronous channel with a peculiar behavior. Analogous to a regular Sync channel, this connector allows pairs of write and take operations to succeed on its two "ends" a and b, only simultaneously. The peculiar behavior of this "synchronous channel" is that regardless of what value is written to a, it always produces the same constant value (in this case the number 5) through b.

11.12 Ordering

The connector in Figure 6.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. The filters of all channels are the wild-card pattern *. Let us consider the behavior of this connector, assuming a number of eager producers 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 the write operations performed on their respective nodes, irrespective of the actual data items that they write. In other words, we may

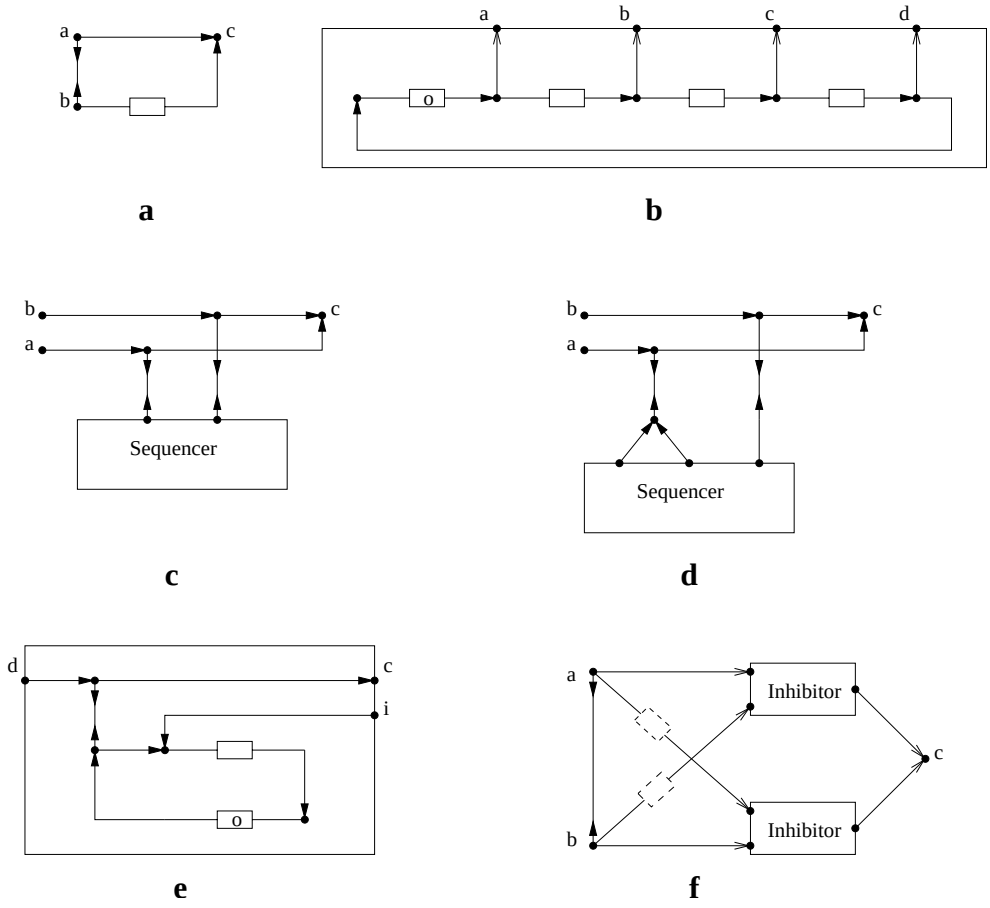


Figure 6: Connectors for more complex coordination

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.

12 Expressiveness

The producers and consumers connected to the nodes `a`, `b`, and `c` of the connector in Figure 6.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 $P\epsilon\omega$. It is natural at this point to wonder about the expressiveness of the composition paradigm of $P\epsilon\omega$, i.e., Given a (small) set of primitive channel types, what coordination patterns can be implemented in $P\epsilon\omega$ by composition of such channel types?

In this section we demonstrate, by examples, that $P\epsilon\omega$ 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`, `AsyncDrain`, an asynchronous channel with the bounded capacity of 1 (e.g., `FIFO1`), and a lossy version of the latter (e.g., `ShiftFIFO1` or `LossyFIFO1`).

12.1 Sequencer

Consider the connector in Figure 6.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 6.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 6.c is another connector for the coordination pattern expressed as $c = (ab)*$. However, there is a subtle difference between the connectors in Figures 6.a and c: the one in Figure 6.a never allows a `write` to `a` succeed without a matching `write` to `b`, whereas the one in Figure 6.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 6.d corresponds to the meta-regular expression $c = (aab)*$. Figures 6.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.

12.2 Inhibitor

The connector in Figure 6.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.

Our "or" selector can now be constructed out of two inhibitors and two `LossyFIFO1` channels, plus some other connector for non-deterministic choice. The connector in Figure 6.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 non-deterministic 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, $c = (a|b)*$.

13 Examples of Connector Construction

In this section we examine a number of non-trivial connectors through which we demonstrate how some useful and more complex connectors can be composed out of simpler ones.

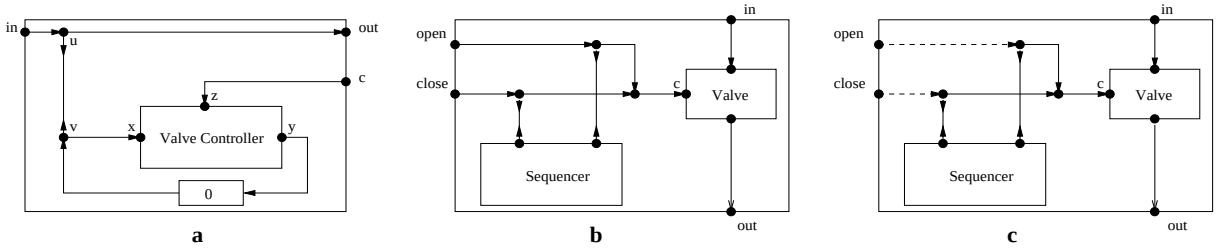


Figure 7: Valves

13.1 Valve

The inhibitor connector in Figure 6.e (Section 12.2) has an irreversible behavior: once a value is written to `i` it inhibits the flow of values from `d` to `c` permanently, and there is no way to "unblock" this flow. The connector in Figure 7.a looks and behaves very much like such an inhibitor, except that successive values written to `c` alternately inhibit and resume the flow from `in` to `out` nodes. We call the connector in Figure 7.a a valve, and say successive values written to node `c` effectively close and open this valve, regulating the flow from `in` to `out`.

The difference between the inhibitor in Figure 6.e and our valve is that we have a new "Valve Controller" connector in place of the `FIFO1` channel of Figure 6.e. Whereas a value entering the `FIFO1` channel permanently blocks the cycling of the token value, preventing the flow, we expect the valve controller to alternately disable and enable the flow from `x` to `y`, inhibiting and allowing the cycling of the token and the flow from `in` to `out`. The exact details of how such a connector is constructed is irrelevant at this point; we will consider two versions of such a valve controller in Sections 13.4 and 13.5. For our purposes, it suffices to imagine that the connector labeled "Valve Controller" in Figure 7.a behaves similar to its corresponding `FIFO1` channel in Figure 6.e, except that values written to `c` alternately "deposit" and "remove" the data item that blocks the cycling of the token contained in the `FIFO1` channel `yv`.

The behavior of the valve is, thus, as follows. The valve is initially open. While open, all values flow from `in` to `out`. The arrival of the first value on `c` closes the valve. While closed, no value flows from `in` to `out`. The arrival of the subsequent values on `c` alternately open and close the valve.

13.2 Dual Control Valve

The functionality of the valve shown in Figure 7.a is very useful, but the exact way in which this valve is opened and closed can be made more flexible by dedicating a separate control for each of these two states. We refer to a valve connector that exposes two separate control nodes, one for opening and one for closing the valve, as a *dual control valve*.

A dual control valve can be composed out of a binary sequencer and a simple valve, as shown in Figure 7.b. The box labeled "Valve" in Figure 7.b is exactly the valve shown in Figure 7.a. The box labeled "Sequencer" in Figure 7.b is the same as the one shown in Figure 6.c. Each of the two nodes of the sequencer in Figure 7.b is connected to a `SyncDrain` channel and all other channels in this figure are `Sync` channels.

It is not difficult to see that the sequencer in Figure 7.b orders the control values written to the exposed nodes `open` and `close` such that the values that arrive on node `c` of the valve represent tokens to alternately close and open the valve. Observe that this version of our valve strictly performs every single open and close request, and strictly orders them alternately so that they make sense: an open valve cannot be opened, so such an action is delayed until after the valve is closed; and the same goes for closing a closed valve.

13.3 Dual Control Valve with Over-ride

The strictness of the dual control valve in Figure 7.b in performing every open and close operation may not be appropriate in some situations. We may wish to have a dual control valve that interprets opening of an open valve and closing of a closed valve as no-operations. Such a valve, in effect, over-rides the requested operation if it agrees with its status. We refer to this version of our connector as a *dual control valve with over-ride*.

Figure 7.c shows our dual control valve with over-ride. Compared to the dual control valve in Figure 7.b, we see that the only difference is that the two `Sync` channels connected to `open` and `close` nodes are replaced with `LossySync` channels, represented as dashed arrows. The effect of these `LossySync` channels is that if a (close or open) value is not expected by the sequencer, the value is lost. Thus, once the valve is open, the sequencer allows only a close token to come through; all open tokens will be lost. Likewise, once the valve is closed, the sequencer expects only an open token, and the `LossySync` channel loses all other close tokens.

```
1   FlusherComponent(chantype t)
2   {
3     ⟨a, g⟩ = create(Sync)
4     ⟨e, b⟩ = create(Sync)
5     ⟨c, f⟩ = create(Sync)
6     connect(g)
7     connect(e)
8     connect(f)
9     run Flusher(t, g, e, f)
10    return ⟨a, b, c⟩
11  }
```

Table 7: The flusher component

13.4 Dynamic Valve Controller

Clearly, the box labeled "Valve Controller" in Figure 7.a cannot be a simple channel, because it exposes three channel ends (labeled `x`, `y`, and `z`). This leaves two possibilities: it can be an instance of either a component, or a connector. In this section we consider the valve controller as a component instance, because its functionality presents an opportunity to demonstrate the utility of some of the dynamic aspects of $P\epsilon\omega$. We see in Section 13.5 that this same functionality can also be realized as a static connector composed of channels.

Table 7 shows the pseudo-code of a simple component that we can use to instantiate our valve controller. An instance of this component requires an actual parameter that identifies a channel type. Specifically, our valve controller can be instantiated as `FlusherComponent(FIFO1)`. The code in Table 7, then, creates three `Sync` channels; connects and passes one end of each of these channels, together with the parameter `FIFO1`, to the `Flusher` function; and returns the other ends of the three `Sync` channels. Observe that the `run` keyword on line 9 indicates that the `Flusher` function starts executing as a separate active entity (e.g., as a thread or a process) within the same component instance. Thus, although `FlusherComponent` terminates and returns, an active entity will remain in this component instance executing the code of the `Flusher` function.

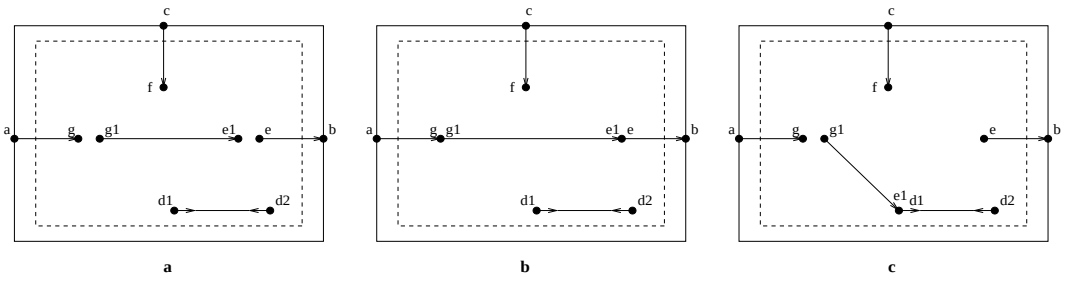


Figure 8: Flusher configurations

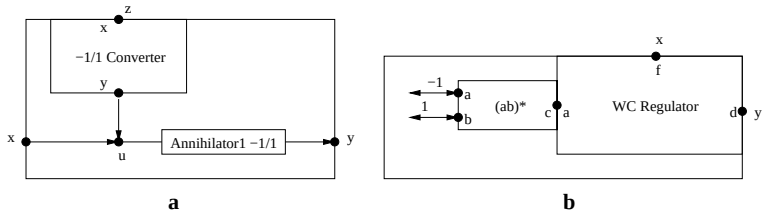
Table 8 shows the code of the `Flusher` function. Figure 8.a shows the configuration of the channels created in this component instance right before the while loop on line 5 in the `Flusher` function. There are five channels involved: the three channels created in `FlusherComponent` (only one end of each of which is known in the `Flusher` function, as indicated by the dashed boxes in Figure 8), and the two channels created in the `Flusher` function itself. Statements 6 and 7 join the ends of the `FIFO1` channels such that there is a connection between `a` and `b`. Statement 8 causes the active entity executing the `Flusher` function to suspend until it can consume a value through `f`. Meanwhile, the connection from

```
1   Flusher(chantype t, outp g, inp e, inp f)
2   {
3     ⟨d1, d2⟩ = create(AsyncDrain)
4     ⟨g1, e1⟩ = create(t)
5     while (true) do
6       join(g, g1)
7       join(e, e1)
8       take(f)
9       g1 = split(g, e)
10      e1 = split(e, g1)
11      connect(d1)
12      connect(e)
13      connect(g)
14      join(d1, e1)
15      wait(g1, empty)
16      e1 = split(e1, g1)
17    done
18  }
```

Table 8: The flusher function

`a` to `b` behaves exactly the same as a `FIFO1` channel. This configuration is shown in Figure 8.b. When a value is written to `c`, the active entity executing the `Flusher` function executes the code on lines 9 and 10, which splits the connections in Figure 8.b and reverts the configuration back to the one shown in Figure 8.a. The `connect` statements on lines 11 through 13 ensure that their respective channel ends are connected to the component instance, so that the subsequent `join` operations can succeed. Next, the join on line 14 changes the configuration to the one shown in Figure 8.c, at which point the active entity executing the `Flusher` function waits (on line 15) until the `FIFO1` channel is flushed empty through its connection with the `AsyncDrain` channel. Once this happens, the split on line 16 reverts the configuration back to the one in Figure 8.a, and the next iteration begins.

We can see that in each iteration, the arrival of a value on `c` flushes the contents of the `FIFO1` buffer that connects `a` to `b`.



a **b**

Figure 9: Static valve controller and -1/1 converter

13.5 Static Valve Controller

The same functionality of the box labeled "Valve Controller" in Figure 7.a, as implemented by the component described in Section 13.4, can also be implemented as a connector with a static topology. Such a static valve controller is shown in Figure 9.a. It consists of a connector labeled "-1/1 Converter" plus an annihilator channel, `uv`, and two `Sync` channels, `xu` and `yu`. The annihilator channel (see Section 7.9) `uy` has a bounded capacity of 1 and treats the values -1 and 1 as each other's anti-values: if its buffer

contains a -1 (or 1) value, then writing a 1 (or -1) value succeeds on this channel, the two (anti-)values annihilate each other, flushing the buffer of the channel to empty.

A node or channel end may be internally known to a connector under a different name that it is known to other components or connectors. This is shown in Figure 9 by having different names for the same node inside and outside of a box that represents a connector.

The function of the box labeled "-1/1 Converter" is to produce through `y` an alternating sequence of -1 and 1 values, every time a value is written to `z`. This connector is described in Section 13.6. Assuming that the annihilator channel is initially empty, and a -1 or 1 value is never written to its exposed channel end `x` nor taken through its exposed channel end `y`, it is not difficult to see that this connector allows values written to its exposed channel end `x` flow through its exposed channel end `y`, until a value is written to `z`, which injects a -1 through `u` into the annihilator channel. As soon as this value enters the buffer of the annihilator, the flow from the controller's exposed channel ends `x` to `y` stops. Because this value is never consumed through `y`, it remains in the buffer of the annihilator until another value is written to `z`, injecting a 1 through `u` into the annihilator channel. At this point, the 1 can enter the annihilator's buffer and erase the -1 value, allowing the flow from `x` to `y` to resume.

13.6 A -1/1 Converter

It is easy to construct the -1/1 converter used in Figure 7.a as a connector composed out of some of the other connectors we have already discussed earlier in this paper. Figure 7.b shows this connector constructed out of two `AsyncSpout` channels, a binary sequencer, and a write-cue regulator. The binary sequencer, labeled "(ab)*" in Figure 7.b, is either the one shown in Figure 6.a (Section 11.12) or Figure 6.c (Section 12.1). The write-cue regulator is the connector shown in Figure 4.e (Section 11.5).

Every write to `x` allows a value to pass within the WC Regulator from its node `a` to its node `d`, which is exposed as node `y` to the external world. The node known as `a` in the WC Regulator is the same node known as `c` in the binary sequencer. The sequence of values that this sequencer produces through this node consists of the alternating values -1 and 1. Therefore, every write to the node `x` of this connector produces a -1 or 1 through its node `y`.

14 Modeling Other Models

The variety of channels that can be used in $P\epsilon\omega$ makes it easy to construct other coordination models. For instance, we consider how Manifold and a Linda-like shared tuple space model can be modeled in $P\epsilon\omega$ in the following two subsections.

14.1 Manifold

The constructs in the coordination language Manifold [1, 8] 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 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 Manifold 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. Raising 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.

14.2 Tuple Spaces

A Linda-like tuple space can be constructed in $P\epsilon\omega$ using a bag channel type. The identity of a tuple space will be the pair of channel ends that identify its corresponding bag channel. Each Linda operation then consists of a function that accepts this tuple space identity as one of its parameters and uses it to refer to the appropriate end of its bag channel.

The following algorithm shows the example of a Linda-like read operation. This algorithm takes the identifier of the tuple space, ⟨`tsrc, tsnk`⟩, a variable, `v`, and a pattern, and returns in `v` a copy of a value in the tuple space that matches with that pattern.

```
Lrd(⟨outp tsrc, inp tsnk⟩, var v, pattern pat)
{
  connect(tsnk)
  read(tsnk, v, pat)
  disconnect(tsnk)
}
```

Table 9: Linda read operation

This function first suspends its caller until it is granted the (exclusive) connection to the sink of the bag channel representing the tuple space. Next, it performs a `read` operation, and subsequently disconnects from the sink channel end.

15 Conclusion

$P\epsilon\omega$ 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 $P\epsilon\omega$. So long as these channels comply with a non-restrictive set of requirements defined by $P\epsilon\omega$, the semantics of $P\epsilon\omega$ operations, especially its composition, is independent of the specific behavior of channels. These requirements define the generic aspects of the behavior of channels that $P\epsilon\omega$ cares about, ignoring the details of their specific behavior. The composition of channels into complex connectors in $P\epsilon\omega$ relates their specific semantics to each other in a manner that is independent of their specific semantics.

The semantics of composition of connectors in $P\epsilon\omega$ 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. $P\epsilon\omega$ connector diagrams can be used as the "glue code" that supports and coordinates inter-component communication in a component based system. As such, drawing $P\epsilon\omega$ connector diagrams constitutes a visual programming paradigm for coordination and component composition.

Connector composition in $P\epsilon\omega$ 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 $P\epsilon\omega$ connectors composed out of a small set of only five primitive channel types.

Our on-going work on $P\epsilon\omega$ 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 $P\epsilon\omega$ to support composition of component based software systems in Java, and the development of logics for reasoning about connectors.

16 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 $P\epsilon\omega$. I am also thankful to J. Rutten for his keen interest in $P\epsilon\omega$ and his inspiring preliminary work on a coalgebraic formal semantics for it. The members of the bi-weekly ACG seminar of J. de Bakker at CWI patiently heard and enthusiastically discussed various aspects of $P\epsilon\omega$ in three different afternoon sessions in 2001. I am grateful for their attention and the creative influence of these discussions.

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. Manifold version 2: Language reference manual. Technical report, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.

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

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

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

[6] Farhad Arbab. Coordination of mobile components. In Ugo Montanari and Vladimiro Sassone, editors, *Electronic Notes in Theoretical Computer Science*, volume 54. Elsevier Science Publishers, 2001.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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