

Coordination for Component Composition

Farhad Arbab^{1,2}

*Center for Mathematics and Computer Science (CWI)
Kruislaan 413
1098 SJ Amsterdam
The Netherlands*

*Leiden Institute for Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands*

Abstract

Composition of systems out of autonomous subsystems pivots on coordination concerns that center on interaction. Interaction has been studied as an inseparable concern in concurrency theory. Curiously, however, interaction has not been seriously considered as a first-class concept in *constructive models of computation*. The coordination language Reo provides a powerful and expressive model for flexible composition of behavior through interaction. Reo serves as a good example of a constructive model of computation that treats interaction as a (in fact, *the only*) first-class concept. It uniquely focuses on the compositional construction of connectors that enable and coordinate the interactions among the constituents in a concurrent system, without their knowledge. We show how Reo allows complex behavior in a system to emerge as a composition of primitive interactions.

Keywords: Components, Composition, Connectors, Reo, Abstract Behavior Types, Exogenous Coordination.

1 Introduction

The desire to compose running systems by gluing together existing pieces of software and subsystems as reusable components, and to verify that the resulting system behaves as expected sits at the core of component based software engineering. Composition of web services makes this core concern even more challenging: service oriented computing requires coordinated composition (also referred to as “choreography” or “orchestration”) of the externally observable behavior of separate pieces of software whose actual code cannot be composed and must remain within the purview of independent autonomous organizations.

¹ This work was conducted with partial support from the Dutch Science Organization (NWO) for the C-Quattro project number 612.000.316, and the SENTER funding for the CIM III project.

² Email: farhad@cw.nl

Software composition has been a concern since the inception of programming. Function calls, method invocation, remote procedure calls, and their variants comprise the mechanisms used to compose software in most contemporary models. These mechanisms are very effective for *composing algorithms*. To tackle dynamic *composition of behavior* by orchestrating the interactions among independent distributed subsystems or services, requires new models for software composition centered on *interaction* as a first-class concept. Various aspects of interaction protocols have been studied in concurrency theory. Curiously, however, interaction has not been seriously considered as a first-class concept in *constructive models of computation*.

Contemporary models of concurrency, such as CSP [22], CCS [25], the π -calculus [26,29], process algebras [10,11,21], and the actor model [1], predominantly treat interaction as a secondary or derived concept. Process calculi, for instance, are models for constructing processes. They offer operators for composing atomic processes or primitive actions into more complex processes. Interaction ensues only as a consequence of the unfolding of the behavior of the processes involved in a concurrent system. For example, as a process p unfolds and performs its actions, one of its primitive actions, such as a send, collides with a compatible primitive action, such as a receive, performed by another process q . It is this collision of actions that forms an interaction. Whether this collision occurs by dumb luck, divine intervention, or intelligent design, is irrelevant. A split-second earlier or later, perhaps in a different run, the same two actions could have collided with other actions of other processes, yielding entirely different interactions. Actions and their composition have explicit constructs used to define a system. Interaction is ephemeral and implicit, and plays no structural role in the construction of a system. Other contemporary models for software composition, such as the object oriented paradigm or the actor model, fair no better than process calculi in this regard.

A constructive model of computation wherein interaction is a first-class concept must offer (1) primitive interactions; and (2) rules of composition for combining (primitive) interactions into more complex interactions, without the need to specify (the actions of) the actors involved.

The coordination language Reo serves as a good example of a constructive model of interaction. In this paper we briefly describe Reo and demonstrate that it provides a powerful and expressive model for flexible composition of behavior through interaction. Reo uniquely focuses on the compositional construction of connectors that enable and coordinate the interactions among the constituents in a concurrent system, without their knowledge. Reo shows how complex behavior in a system can emerge as a composition of primitive interactions.

2 Exogenous Coordination

Exogenous coordination [4] means coordination from outside and refers to the ability, in a model or language, to coordinate the behavior of black-box entities, without their knowledge, from outside of those entities. This is an essential property for a

component composition model to have because it allows building systems with very different emergent behavior out of the exact same components, simply by composing them differently. A vivid example of the significance of exogenous coordination appears in [6], with two instances of the classical dining philosophers problem. Different connectors can exogenously impose different coordination protocols on the same components (e.g., philosophers and chopsticks) to yield different composed systems that exhibit different emergent system behavior. In the case of the dining philosophers, for instance, the possibility of deadlock as an emergent behavior can be eliminated simply by composing the same components differently.

Unix pipes and filters serve as an example of how independent executable pieces of software can be exogenously coordinated into a composed system. Alas, the limited flexibility of this model restricts its expressiveness to but the simplest forms of (pipeline) composition. Classical dataflow models, dataflow-like networks and calculi such as [14,15], [18], [24], Kahn networks [23], and Petri nets each incorporates specific coordination constructs that offer more flexibility. In the context of software composition, these models have shortcomings in at least two significant areas. First, they do not allow mixing synchrony and asynchrony in behavioral definitions. Second, they support, at best, only very rudimentary forms of exogenous coordination.

As an example, suppose we have three components, C , D , and T , as in Figure 1.a. They are all black-box components: we know nothing about what they are made of or how they work internally. They may be made out of hardware, software, or some combination of the two. We can make no assumptions about the language or model used to construct these components. Specifically, they neither provide an interface of methods to call, nor make any method calls to interact with their environment.

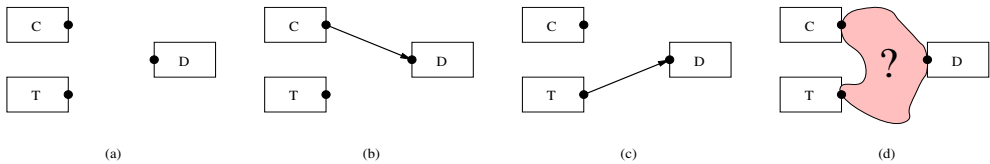


Fig. 1. Three components and their various compositions

The only thing we know about C is what we can externally observe of its behavior. It has a single port of interaction with its environment, through which it periodically outputs some string of characters. Of course, for the output to take place, (an entity in) the environment of C must be prepared to accept its output. Assuming an ideally cooperative environment (i.e., always ready to take it whenever C attempts to output its string), C produces a string approximately every 15 seconds, with the tolerance margin of ϵ . The actual content of the strings produced by C is the current time; so C is a clock.

The only thing we know about D is that it has a single input port, through which it consumes strings and displays them on its accompanying monitor for approximately 30 seconds. The “processing time” of D is negligible for our purposes.

We observe that T behaves very much the same as C , except that its tolerance margin is δ and the content of its output strings convey the current temperature.

We can construct a few systems out of these components, the simplest ones involving a direct connection, e.g., between C and D . Because we cannot alter any of these components, we must make the connection from outside. The simplest connector we can use to compose C and D is what we call a synchronous channel, as in Figure 1.b. Like a buffer-less Unix pipe, a synchronous channel is a medium of communication with two ends. Through one of its ends, it accepts input, and through the other, it dispenses it. We call it “synchronous” because it synchronizes the pair of input and output operations at its opposite ends: the two operations are suspended as necessary to ensure that they succeed together atomically.

If we connect C to D using a synchronous channel whose transfer and synchronization time is negligibly small (compared to the period of C), we obtain a composed system that displays the current time, updated approximately every 30 seconds. Similarly, we can construct another system out of T and D connected by a synchronous channel, as in Figure 1.c, to display the current temperature, updated approximately every 30 seconds.

In order to build a system, similar to what one finds on the top of some bank buildings, that alternately displays the current time and temperature, we have all the functional elements that we need in C , D , and T . What we need is a connector to compose them together as in Figure 1.d. This connector must have a more complex behavior than that of a synchronous channel used in the previous compositions: not only it must facilitate the data exchanges among these three components, but it also needs to enforce the coordination protocol that implements the desired alternating behavior. Because the internals of the components cannot be changed, such a connector would have to impose its coordination protocol “from the outside” of the components, which illustrates what we mean by exogenous coordination.

Obviously, such a connector, as well as other even more sophisticated ones, can be developed as programs in any modern programming language; their Turing completeness ensures that. However, it is interesting to ponder if there is a better, higher-level alternative to programming such connectors from scratch. Synchronization and coordination protocols are notoriously complex concurrent programs, and adding provisions to enable them to cope with mobility in distributed environments makes conventional programming models and languages grossly inadequate for their development. There is enough commonality of purpose (facilitating data exchange and exogenous coordination) among such connectors to warrant considering a special connector specification model and a special language for their development. To the extent that they merely connect and coordinate and lack application-specific functionality, each such connector can be generically designed and reused to compose widely different sets of components into entirely different systems.

What would a special purpose connector specification model look like? Can connectors be reused not just to compose components into (sub)systems, but also to compose more complex connectors? What composition operators are necessary and sufficient to allow connector composition? Is there a set of primitive connectors out of which “all interesting or useful” connectors can be constructed by those connector composition operators? How can one characterize interesting and useful

in this context?

In the rest of this paper, we address these questions in the context of a concrete model, Reo, and show how it serves as a language for compositional construction of reusable coordinating component connectors.

3 Reo

Reo is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones [5]. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. The emphasis in Reo is on connectors, their behavior, and their composition, not on the entities that connect, communicate, and cooperate through them. The behavior of every connector in Reo imposes a specific coordination pattern on the entities that perform normal I/O operations through that connector, without the knowledge of those entities. This makes Reo a powerful “glue language” for compositional construction of connectors to combine component instances into a software system and exogenously orchestrate their mutual interactions. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of primitive channels.

Component instances, as well as channels, can be mobile in Reo. Logical mobility of channel ends in Reo allows dynamic reconfiguration of connectors, even while they are being used by component instances. In this respect, Reo resembles dynamically reconfigurable generalized Kahn networks, as in IWIM [4] and Manifold [12].

Broy’s work on timed dataflow channels [14,15] is perhaps closest to Reo. Here, components are functions that transform input data streams to output data streams, which represent their interconnecting FIFO channels. The only notion of “time” in this model arises out of sporadic “tick” marks intermixed with the data within the same streams. In contrast to Reo, streams/channels cannot be directly connected or composed together in this model: they can exist only between two components, which use the tick marks to synchronize their various input and output streams. This gives a functional (i.e., uni-directional transformation) flavor to the model. In contrast, Reo circuits are *relational* (i.e., bi-directional constraints). Furthermore, Reo has a more general notion of channels, allows inherently dynamic topologies, and its notion of channel/connector composition allows, among other things, compositions involving an expressive mix of synchrony and asynchrony.

3.1 Components

Reo regards a component instance as a black-box entity. Reo assumes that every component instance contains one or more active entities whose only means of communication with other entities outside of that component instance is through regular input/output of passive data. Specifically, I/O of passive data precludes transfer of control, method invocation, and targeted messages. A component instance performs its I/O operations following its own timing and logic, independently of the others. However, for such an I/O operation to succeed, the environment of the

component instance must offer a suitable matching I/O operation as well. Thus, when a component instance attempts to write some data item, its output operation blocks until its environment accepts to take that data item; when a component instance attempts to read, its input operation blocks until its environment offers it a data item. Of course, a component instance may specify a time-out for each I/O operation that it attempts to perform, to allow it to retract its offered I/O, rather than wait indefinitely for its environment to match it.

A Unix process, for instance, qualifies as a component instance: it contains one or more threads of control which may even run in parallel on different physical processors, and its file descriptors qualify as ports. A component instance may itself consist of a collection of other component instances, perhaps running in a distributed environment. Thus, by identifying their relevant ports through which they exchange data with their environment, entire systems can be viewed and used as component instances, abstracting away their internal details of operation, structure, geography, and implementation.

This notion of component is different than what most other models consider as their components. Our components are intrinsically active, do not issue, and do not accept method calls. However, any abstraction, X , offered as a “component” by an alternative contemporary model (e.g., ArchJava [3,2], JavaBeans [20], CORBA [17], COM+ [16], etc.) can always be wrapped in a thin layer of *adapter* code to yield a component in our model. This adapter layer (whose code can even be mechanically generated) creates an active entity, if necessary, and acts as an intermediary that converts the passive input/output messages exchanged between the component and its environment, to the method calls expected and issued by its encapsulated X .

3.2 Channels

Reo defines a number of operations for components to (dynamically) compose, connect to, and perform I/O through connectors. Atomic connectors are *channels*. The notion of channel in Reo is far more general than its common interpretation.

Reo defines a channel as a primitive communication medium with its own unique identity, that has exactly two ends together with a constraint that inter-relates the timing and the content of the I/O operations through these ends. There are two types of channel ends: *source* end through which data enters and *sink* end through which data leaves a channel. A channel must support a certain set of primitive operations, such as I/O, on its ends; beyond that, Reo places no restriction on the behavior of a channel. Reo does not even insist that a channel must have one source and one sink; it also admits channels with two sources or two sinks. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc.

3.2.1 A Sample of Channels

Figure 2 shows a sample set of primitive channel types and the graphical symbols we use to represent them.




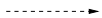


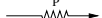

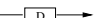
 Synchronous	 Synchronous Drain	 FIFO
 Lossy Synchronous	 Synchronous Spout	 FIFO1
 Filter(P)	 Asynchronous Drain	 FIFO1(D)

Fig. 2. A set of primitive channel types and their graphical symbols

A synchronous channel, **Sync**, graphically represented as a solid arrow, has a source- and a sink-end. This channel synchronizes the success of the two I/O operations on its two ends. In other words, it blocks a write operation on its source end or a take operation on its sink end, as necessary, to ensure that these two operations succeed atomically.

SyncDrain is a synchronous channel with two source ends; it has no sink end. This means no one can ever take any data out of this channel. Therefore, all data entered into this channel are lost. **SyncDrain** is a synchronous channel in exactly the same sense as a **Sync** channel: it synchronizes the two I/O operations on its ends. In this case they must both be write operations, and **SyncDrain** blocks either of the two, as necessary, to ensure that they succeed atomically.

FIFO is an asynchronous channel with a source end and a sink end with an unbounded buffer to contain data. Its buffer is initially empty. With an unbounded buffer, a write operation on its source end always succeeds, placing its data in the buffer. With a non-empty buffer, a take on the sink end of this channel succeeds and removes the oldest data item in the buffer. When the buffer is empty, a take operation on the sink end of this channel blocks, waiting for the status of the buffer to change.

LossySync is a synchronous channel with a behavior very similar to that of the **Sync** channel. Just as for a **Sync** channel, a take operation on the sink end of a **LossySync** blocks until a write is performed on its source end. Unlike the case of the **Sync** channel, all write operations on the source end of a **LossySync** immediately succeed: if there is a pending take on its sink end, then the written data item is transferred; otherwise, the write operation succeeds, but the written data item is lost.

A synchronous spout, **SyncSpout**, disposes data items out of its two ends only synchronously. The actual values it produces through its ends are nondeterministic.

FIFO1 is an asynchronous channel with a source end and a sink end and a bounded buffer with the capacity to contain at most 1 data item. Its buffer is initially empty. With an empty buffer, a write operation on its source end succeeds and fills the buffer. With a non-empty buffer, a take on the sink end of this channel succeeds and removes the data. Otherwise, I/O operations block waiting for the status of the buffer to change. **FIFO1(D)** is a variant of the **FIFO1** channel whose buffer initially contains the data item **D**.

A **Filter(P)** channel is a synchronous channel with a source and a sink end that takes a pattern **P** as parameter upon its creation. It behaves like a **Sync** channel, except that only those data items that match the pattern **P** can actually

pass through it; others are always accepted by its source end, but are immediately lost.

An asynchronous drain **AsynchDrain** is the dual of a **SyncDrain**: it allows the two write operations on its two ends to succeed only one at a time, i.e., never simultaneously together.

3.3 Nodes

A node is an important concept in Reo. Not to be confused with a location or a component, a node is a logical construct representing the fundamental topological property of coincidence of a set of channel ends, which has specific implications on the flow of data among and through those channel ends.

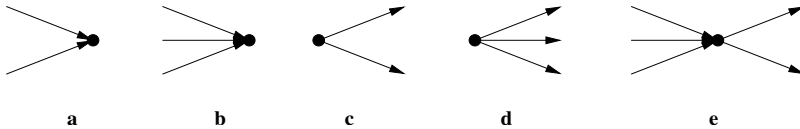


Fig. 3. Sink, Source, and Mixed nodes

The set of channel ends coincident on a node A is disjointly partitioned into the sets $Src(A)$ and $Snk(A)$, denoting the sets of source and sink channel ends that coincide on A , respectively. A node A is called a *source node* if $Src(A) \neq \emptyset \wedge Snk(A) = \emptyset$. Analogously, A is called a *sink node* if $Src(A) = \emptyset \wedge Snk(A) \neq \emptyset$. A node A is called a *mixed node* if $Src(A) \neq \emptyset \wedge Snk(A) \neq \emptyset$. Figures 3.a and b show sink nodes with, respectively, two and three coincident channel ends. Figures 3.c and d show source nodes with, respectively, two and three coincident channel ends. Figure 3.e shows a mixed node where three sink and two source channel ends coincide.

The expressive power of Reo stems from the behavior of its nodes. Reo provides operations that enable components to connect to and perform I/O on source and sink nodes only; components cannot connect to, read from, or write to mixed nodes. At most one component can be connected to a (source or sink) node at a time. A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items from a sink node that it is connected to through destructive (take) and non-destructive (read) input operations. A take operation succeeds only if at least one of the (sink) channel-ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node is a self-contained “pumping station” that combines the behavior of a sink node (merger) and a source node (replicator) in an atomic iteration of an endless loop: in every iteration a mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. A data item is suitable for selection

in an iteration, only if it can be accepted by all source channel ends that coincide on the mixed node.

3.4 Connector

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

- (i) Zero or more channel ends coincide on every node.
- (ii) Every channel end coincides on exactly one node.
- (iii) There is an edge between two (not necessarily distinct) nodes if and only if there is a channel one end of which coincides on each of those nodes.

It follows that every channel represents a (simple) connector with two nodes. More complex connectors are constructed in Reo out of simpler ones using its *join* operation. Joining two nodes destroys both nodes and produces a new node on which all of their coincident channel ends coincide.

This single operation allows construction of arbitrarily complex connectors involving any combination of channels picked from an open-ended assortment of user-defined channel types. The semantics of a connector is defined as a composition of the semantics of its (1) constituent channels, and (2) nodes. The semantics of a channel is defined by the user who provides it. Reo defines the semantics of its three types of nodes, as mentioned above.

4 Coordination by Connectors

In this section we show how coordinating connector circuits can be constructed in Reo through channel composition. We start with a few simple examples, followed by a number of non-trivial, generically useful connectors. We then consider a more general version of the time-temperature-display example of Section 2 and build the connector circuit for its coordination.

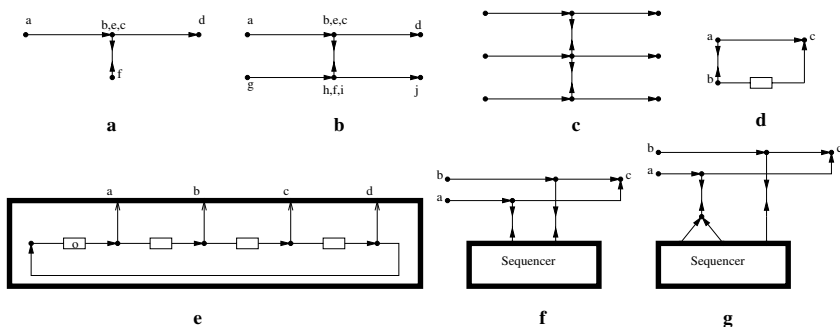


Fig. 4. Examples of connector circuits in Reo

4.1 Write-Cue Regulator

Consider the connector in Figure 4.a, composed out of the three channels **ab**, **cd**, and **ef**. Channels **ab** and **cd** are of type **Sync** and **ef** is of type **SyncDrain**. This connector shows one of the most basic forms of exogenous coordination: the number of data items that flow from **a** to **d** is the same as the number of write operations that succeed on **f**. A component instance connected to **f** can count and regulate the flow of data between the two nodes **a** and **d** by the timing and the number of write operations that 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 consume data items from **b**, nor that its write actions actually regulate this flow. The two entities that communicate through **a** and **d** need not know anything about the fact that they are communicating with each other, nor that the volume of their communication is regulated and/or measured by a third entity at **f**.

4.2 Barrier Synchronizers

We can build on our write-cue regulator to construct a barrier synchronization connector, as in Figure 4.b. The four channels **ab**, **cd**, **gh**, and **ij** are all of type **Sync**. The **SyncDrain** channel **ef** ensures that a data item passes from **a** to **d** only simultaneously with the passing of a data item from **g** to **j** (and vice versa). This simple barrier synchronization connector can be trivially extended to any number of pairs, as shown in Figure 4.c.

4.3 Ordering

The connector in Figure 4.d 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 behavior of this connector can be seen as imposing an order on the flow of the data items written to **a** and **b**, through to **c**: the data items obtained by successive read 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. The coordination pattern imposed by our connector can be summarized 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.

4.4 Sequencer

Consider the connector in Figure 4.e. The enclosing box represents the fact that the details of this connector are abstracted away and it provides only the four nodes of the channel ends **a**, **b**, **c**, and **d** for other entities (connectors and/or component instances) to (in this case) read from. Inside this connector, we have four **Sync**, an initialized **FIFO1**, and three **FIFO1** channels connected together. The initialized **FIFO1** channel is the leftmost one and 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 read operations on the nodes (with channel ends) 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 FIF01 channel pairs, as required.

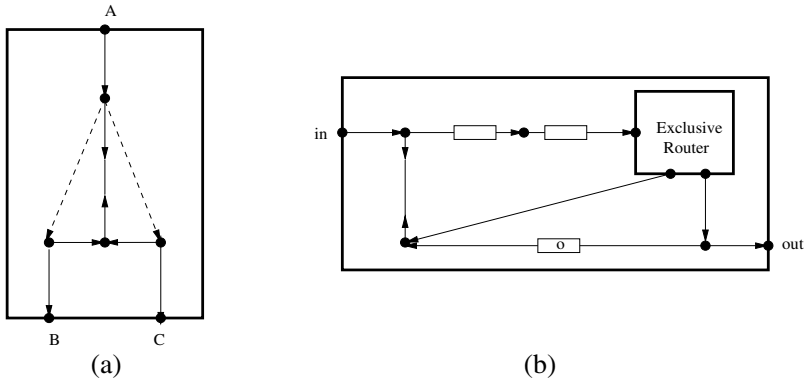


Fig. 5. An exclusive router and a shift-lossy FIF01

4.5 Exclusive Router

Figure 5.a shows the Reo network for an exclusive router connector. A data item arriving at the input port A flows through to only one of the output ports B or C, depending on which one is ready to consume it. If both output ports are prepared to consume a data item, then one is selected nondeterministically. The input data is never replicated to more than one of the output ports. Figure 5.a shows that the exclusive router is composed of two LossySync channels, a SyncDrain channel, and five Sync channels. See [7] for a more formal treatment of the semantics of this connector.

4.6 Shift Lossy FIF01

Figure 5.b shows a Reo network for a connector that behaves as a lossy FIF01 channel with a shift loss-policy. This channel is called shift-lossy FIF01 (ShiftLossyFIF01). It behaves as a normal FIF01 channel, except that if its buffer is full then the arrival of a new data item deletes the existing data item in its buffer, making room for the new arrival. As such, this channel implements a “shift loss-policy” losing the oldest contents in its buffer in favor of the latest arrivals. The connector in Figure 5.b is composed of an exclusive router (shown in Figure 5.a), an initially full FIF01 channel, two initially empty FIF01 channels, and four Sync channels. See [7] for a more formal treatment of the semantics of this connector.

The shift-lossy FIF01 circuit in Figure 5.b is indeed so frequently useful as a connector in construction of more complex circuits, that it makes sense to have a special graphical symbol to designate it as a short-hand. Figure 6 shows a circuit

that uses two instances of our shift-lossy FIF01. The graphical symbol we use to represent this circuit is intentionally similar to that of a regular FIF01 channel, to hint at the similarity of the behavior of these two connectors. As seen in Figure 6, our graphical symbol for a shift-lossy FIF01 “channel” has a half-dashed box instead of the solid box of a regular FIF01 channel: the sink-side half of the box representing the buffer of this channel is dashed, to suggest that it loses the older values to make room for new arrivals, i.e., it shifts to lose.

4.7 Variable

The Reo circuit in Figure 6 implements the behavior of a dataflow variable. It uses two instances of the shift-lossy FIF01 connector shown Figure 5.b, to build a connector with a single input and a single output nodes. Initially, the buffers of its shift-lossy FIF01 channels are empty, so an initial take on its output node suspends for data. Regardless of the status of its buffers, or whether or not data can be dispensed through its output node, every write to its input node always succeeds and resets both of its buffers to contain the new data item. Every time a value is dispensed through its output node, a copy of this value is “cycled back” into its left shift-lossy FIF01 channel. This circuit “remembers” the last value it obtains through its input node, and dispenses copies of this value through its output node as frequently as necessary: i.e., it can be used as a dataflow variable.

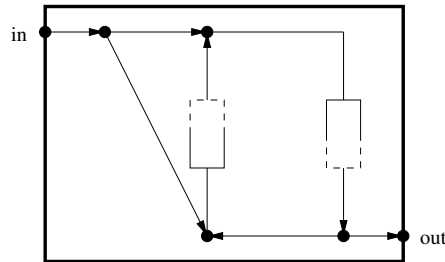


Fig. 6. Dataflow variable

The variable circuit in Figure 6 is also very frequently useful as a connector in construction of more complex circuits. Therefore, it makes sense to have a short-hand graphical symbol to designate it as well. Figure 7 shows 3 instances of our variable used in two connectors. Our symbol for a variable is similar to that for a regular FIF01 channel, except that we use a rounded box to represent its buffer: the rounded box hints at the recycling behavior of the variable circuit, which implements its remembering of the last data item that it obtained or dispensed.

4.8 Time and Temperature Display

Figure 7.a shows a system composed of two components connected via a variable channel presented in Figure 6. The two components labeled *Clock* and *Display* are generalizations of the *C* and *D* components in Figure 1. The *Clock* component periodically produces a text string announcing the current time. The *Display* component periodically reads and consumes a text string and displays it. Unlike

with the *C* and *D* components of Figure 1, we make no assumptions about the periods of **Clock** and **Display**, nor their ratio.

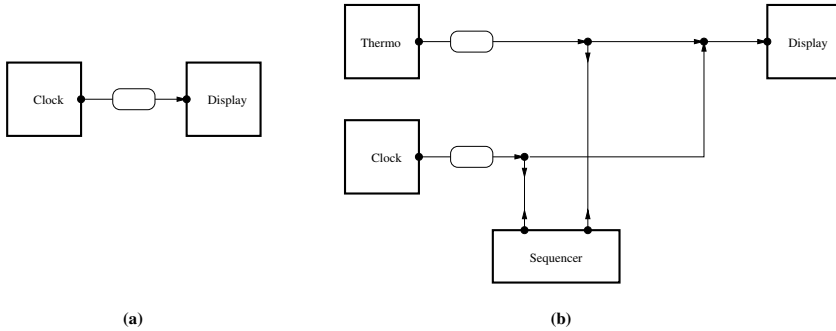


Fig. 7. A Time/Temperature Display system

The variable channel provides temporal decoupling of the clock and the display, while facilitating their communication. Regardless of the state of the display, the clock can always write its current time into the channel, which may lose its old content, if any, to accommodate the new value. As frequently as it wishes, the display can read the current content of the channel, if any, which will be not older than the temporal resolution (i.e., the update cycle) of the clock. If the display’s cycle is faster than that of the clock, the display will read the last value it read, again. If the clock’s cycle is faster than that of the display, it may produce a new value before an older one is consumed by the display. The variable channel allows the new value to override the old. Thus, the system in Figure 7.a periodically displays the current time.

Figure 7.b shows the time-temperature-display system of Figure 1.d, with its proper Reo circuitry. The box labeled **Thermo** in this figure, is a thermometer. Analogous to **Clock**, it is a generalization of the *T* component in Figure 1, with its own arbitrary period. The two variable channels in their connector circuit support communication and temporal decoupling of the clock and the thermometer components from the rest of the system. The input to the display component is regulated by a two-node version of the sequencer connector presented in Figure 4.e. Thus, the system in Figure 7.b alternately displays current time and temperature.

The interesting point about this system is that none of the components involved is aware of the function of the system or of its own collaboration in realizing this “complex” coordinated behavior: the behaviors of the individual components are composed and coordinated exogenously (i.e., from outside of the components) by the Reo connectors to realize this collaborative behavior. Such “ignorant” components are highly generic and reusable, precisely because they are oblivious to whether they are used in a system like in Figure 7.a, or to build a system with a more complex coordination scheme as in Figure 7.b.

5 Expressiveness

Figure 4.f 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 4.f is another connector for the coordination pattern $c = (ab)^*$, although there is a subtle difference between the behavior of this connector and the one in Figure 4.d. See [5] for more detail.

It takes little effort to see that the connector in Figure 4.g corresponds to the meta-regular expression $c = (aab)^*$. Figures 4.f and g show how easily we can construct connectors that exogenously impose coordination patterns corresponding to the Kleene-closure of any “meta-word” made up of atoms that stand for I/O operations, using a sequencer of the appropriate size.

Channel composition in Reo is a very powerful mechanism for construction of connectors. For instance, exogenous coordination patterns that can be expressed as (meta-level) regular expressions over I/O operations performed by component instances can be composed in Reo out of a small set of only five primitive channel types³. A Turing machine consists of a finite state automaton for its control, and an unbounded tape. Since an unbounded tape can be simulated by two unbounded FIFO channels, adding FIFO to the above set of channel types makes channel composition in Reo Turing complete.

6 Abstract Behavior Types

The notion of Abstract Behavior Type (ABT) is introduced in [6] and proposed as a proper foundation model for components and their composition. The ABT model supports a much looser coupling than is possible with the operational interfaces of Abstract Data Types (ADT), and is inherently amenable to exogenous coordination. Both of these are highly desirable, if not essential, properties for models of component behavior and composition of interactions.

An ABT defines an abstract behavior as a constraint among the observable input/output that occur through a set of “contact points” (e.g., ports of a component instance) without specifying any detail about the operations that may be used to implement such behavior, or the data types those operations may manipulate for the realization of that behavior. This definition parallels that of an ADT, which abstracts away from the instructions and the data structures that may be used to implement the operational interface it defines for a data type. In contrast, an ABT defines a behavior in terms of a constraint on the observable input/output of an entity, without saying anything about how it can be realized.

There are several different ways to formalize the concept of ABT. For instance, *constraint automata* [7] offer an operational model of ABTs. In principle, process

³ In fact, Reo more naturally models infinite behavior through infinite streams (see Section 6). As such, composition of this set of primitive channels actually yields the equivalent of ω -regular expressions, rather than (finite) regular expressions. Therefore, for instance, the behavior of the connector in Figure 4.g, more accurately corresponds to the meta-regular expression $c = (aab)^\omega$, rather than $c = (aab)^*$.

calculi, Petri nets, logic expressions, or labeled transition systems can also be used to describe transformations of input to output sequences of observables. In order to emphasize Reo’s perspective of regarding interaction as a constraint, we prefer a formalization that treats an ABT as a relation/constraint, rather than a transformation. The coalgebraic model of ABT based on stream calculus [28], described below, is particularly suited for this purpose.

6.1 Relational View of ABT

The formalization presented in [6] defines an ABT as a (maximal) relation on a set of *timed data streams*, which emphasizes the relational aspect of the ABT model explicitly and abstracts away any hint of an underlying operational semantics of its implementation. This helps to focus on behavior specifications and their composition, rather than on operations that may be used to implement entities that exhibit such behavior and their interactions.

A *stream* (over A) is an infinite sequence of elements of some set A . The set of all streams over A is denoted as A^ω . Streams in $DS = D^\omega$ over a set of (uninterpreted) data items D are called *data streams* and are typically denoted as α, β, γ , etc. Zero-based indices are used to denote the individual elements of a stream, e.g., $\alpha(0), \alpha(1), \alpha(2), \dots$ denote the first, second, third, etc., elements of the stream α . We use the infix “dot” as the stream constructor: $x.\alpha$ denotes a stream whose first element is x and whose second, third, etc. elements are, respectively, the first and its successive elements of the stream α .

Following the conventions of stream calculus [28], the well-known operations of head and tail on streams are called *initial value* and *derivative*: the initial value of a stream α (i.e., its head) is $\alpha(0)$, and its (first) derivative (i.e., its tail) is denoted as α' . Relational operators on streams apply pairwise to their respective elements, e.g., $\alpha \geq \beta$ means $\alpha(0) \geq \beta(0), \alpha(1) \geq \beta(1), \alpha(2) \geq \beta(2), \dots$

Constrained streams in $TS = \mathbb{R}_+^\omega$ over positive real numbers representing moments in time are called *time streams* and are typically denoted as a, b, c , etc. To qualify as a time stream, a stream of real numbers a must be (1) strictly increasing, i.e., the constraint $a < a'$ must hold; and (2) progressive, i.e., for every $N \geq 0$ there must exist an index $n \geq 0$ such that $a(n) > N$.

We use positive real numbers instead of natural numbers to represent time because, as observed in the world of temporal logic [9], real numbers induce the more abstract sense of *dense time* instead of the notion of *discrete time* imposed by natural numbers. Specifically, we sometimes need finitely many steps within any bounded time interval for certain ABT equivalence proofs (see, e.g., [8]). This is clearly not possible with a discrete model of time. The actual values of “time moments” are irrelevant in our ABT model; only their relative order is significant and must be preserved. Using dense time allows us to locally break strict numerical equality (i.e., simultaneity) arbitrarily while preserving the atomicity of events [6].

A *Timed Data Stream* is a twin pair of streams $\langle \alpha, a \rangle$ in $TDS = DS \times TS$ consisting of a data stream $\alpha \in DS$ and a time stream $a \in TS$, with the interpretation that for all $i \geq 0$, the input/output of data item $\alpha(i)$ occurs at “time moment” $a(i)$.

Two timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ are equal if their respective elements are equal, i.e. $\langle \alpha, a \rangle = \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b$.

Formalization of ABT in terms of timed data streams provides a simple yet powerful framework for the formal semantics of Reo. Timed data streams are used to model the flows of data through channel ends.⁴ A channel itself is just a (binary) relation between the two timed data streams associated with its two ends. A more complex connector is simply an n -ary relation among n timed data streams, each representing the flow of data through one of the (non-hidden) n nodes of the connector.

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

$$\langle \alpha, a \rangle \text{ Sync } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b.$$

The equation states that every data item that goes into a **Sync** channel comes out in the exact same order. Furthermore, the arrival and the departure times of each data item are the same: there is no buffer in the channel for a data item to linger on for any length of time.

An asynchronous **FIFO** channel is defined as the relation:

$$\langle \alpha, a \rangle \text{ FIFO } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b.$$

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

An asynchronous **FIFO1** channel is similar to a **FIFO**:

$$\langle \alpha, a \rangle \text{ FIFO1 } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b < a'.$$

Again, everything that goes in comes out in the same order ($\alpha = \beta$). But, for all $i \geq 0$, not only the departure time $b(i)$ of every data item $\alpha(i) = \beta(i)$ is necessarily after its arrival time ($a(i) < b(i)$), but since the channel can contain no more than 1 element, the arrival time $a(i+1)$ of the next data item $\alpha(i+1)$ must be after the departure time $b(i)$ of its preceding element ($a < b < a' \equiv a(i) < b(i) < a(i+1)$, for $i \geq 0$).

A **FIFO1(D)** represents an asynchronous channel with the bounded capacity of 1 filled to contain the data item D as its initial value. The behavior of a **FIFO1(D)** channel is very similar to that of a **FIFO1**:

$$\langle \alpha, a \rangle \text{ FIFO1}(D) \langle \beta, b \rangle \equiv \beta = D.\alpha \wedge b < a < b'.$$

⁴ The infinity of streams naturally models the infinite behavior of perpetual systems. Finite behavior can be modeled in at least three different ways. First, we can allow finite streams as well. Second, it can be modeled as a special case of infinite behavior, e.g., where after a certain time moment, only the special symbol \perp appears as values in all time streams. Although viable, we ignore both of these schemes because they do not add conceptual novelty, yet dealing with the special cases that they involve requires a somewhat more complex formalism. The third way to model finite behavior is to ensure that after a certain point in time, the system has no observable behavior. This is possible with or without finite streams. See footnote 5 in Section 6.4.

This channel produces an output data stream $\beta = D.\alpha$ consisting of the initial data item D followed by the input data stream α of the ABT, and for $i \geq 0$ performs its i^{th} input operation some time between its i^{th} and $i + 1^{\text{st}}$ output operations ($b < a < b'$).

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

$$\langle \alpha, a \rangle \text{ SyncDrain } \langle \beta, b \rangle \equiv a = b.$$

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

$$Rpl(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \beta = \alpha \wedge \gamma = \alpha \wedge b = a \wedge c = a$$

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

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

$$Mrg(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \begin{cases} \alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge Mrg(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\ \beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge Mrg(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) > b(0) \end{cases}$$

6.2 ABT Composition

Because an ABT is a relation, two ABTs can be composed to yield another ABT through a relational composition similar to the join operation in relational databases. This yields a simple, yet powerful formalism for specification of complex behavior as a composition of simpler ones. Composition of simple interaction primitives into non-trivial behavior, such as the Reo circuits in the above examples, can be expressed as ABT composition [6].

The relational (as opposed to functional) nature of our formalism allows a composition of ABTs to mutually influence and constrain each other, yielding their collective behavior, analogous to how a set of constraints in a constraint satisfaction problem resolve into a solution. The use of coinduction as the main definition and proof principle to reason about both data and time streams allows simple compositional construction of ABTs representing many different generic coordination schemes involving combinations of various synchronous and asynchronous primitives that are not present (and not even expressible) in most other models.

A simple example of how a composition of a set of components yields a system that delivers more than the sum of its parts is the computation of the classical Fibonacci series. To assemble an application to deliver this series we actually need only one (instance of an) adder component plus a number of channels.

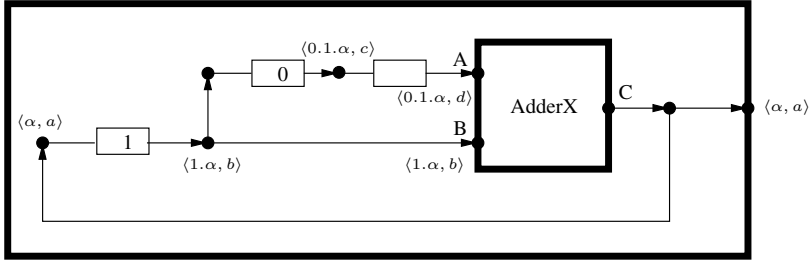


Fig. 8. Fibonacci series in Reo

Figure 8 shows a component (the outermost thick enclosing box) with only one output port (the only exposed node on the right border of the box). This is our application for computing the Fibonacci series. Peeking inside this component, we see how it is made out of an instance of an adder (labeled *AdderX*), a *FIF01(1)*, a *FIF01(0)*, a *FIF01*, and five *Sync* channels. *AdderX* represents a simple adder that repeatedly takes two input values, x and y , respectively through its input ports A and B , and produces a result, z , through its output port C , which is the sum of x and y .

In Section 6.3 we define a few ABTs that formalize some alternatives for the observable behavior of such an adder. Semantically, we can use any one of the adders we define in Section 6.3 in the composition in Figure 8. That is why the box representing the adder in this figure is labeled *AdderX*. However, the extra-semantic behavior of some of these adders makes them unsuitable for the specific circuit in Figure 8. To understand how this circuit is expected to work, suppose *AdderX* has a behavior “compatible” with the circuit. We consider other alternatives in Section 6.4.

Intuitively, as long as the *FIF01(0)* channel is full, nothing can happen: there is no way for the value in *FIF01(1)* to move out. At some point in time, the value in *FIF01(0)* moves into the *FIF01* channel. Thereafter, the *FIF01(0)* channel becomes empty and the two values in the *FIF01(1)* and the *FIF01* channels become available for *AdderX* to consume. The intake of the value in *FIF01(1)* by *AdderX* inserts a copy of the same value into the *FIF01(0)* channel. When *AdderX* is ready to write its computed value out, it suspends waiting for some entity in the environment to accept this value. Transfer of this value to the entity in the environment also inserts a copy of the same value into the now empty *FIF01(1)* channel. At this point we are back to the initial state, but with different values in the buffers of the *FIF01(1)* and the *FIF01(0)* channels.

6.3 Adders

To illustrate the expressiveness of the ABT model and the utility of ABT composition, consider the adder component used in our Fibonacci example in Section 6.2. We define a few of the alternative versions of the behavior for this adder, below, each as a different ABT:

$$\begin{aligned}
\text{Adder1}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\
\gamma(0) &= \alpha(0) + \beta(0) \wedge \\
\exists t : \max(a(0), b(0)) < t < \min(a(1), b(1)) &\wedge c(0) = t \wedge \\
\text{Adder1}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). &
\end{aligned}$$

Adder1 defines the behavior of a component that repeatedly reads a pair of input values from its two input ports, adds them up, and writes the result out on its output port. As such, its output data stream is the pairwise sum of its two input data streams. This component behaves asynchronously in the sense that it can produce each of its output data items with some arbitrary delay after it has read both of its corresponding input data items ($c(0) = t \wedge t > \max(a(0), b(0))$). However, it is obligated to produce each of its output data items before it reads in its next input data item ($t < \min(a(1), b(1))$).

$$\begin{aligned}
\text{Adder2}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\
\gamma(0) &= \alpha(0) + \beta(0) \wedge \\
c(0) &= \max(a(0), b(0)) \wedge \\
\text{Adder2}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). &
\end{aligned}$$

Adder2 behaves very much like *Adder1*, except that it produces the sum of every pair of input values atomically (i.e., synchronously) together with its consuming of its second input value ($c(0) = \max(a(0), b(0))$).

$$\begin{aligned}
\text{Adder3}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\
\gamma(0) &= \alpha(0) + \beta(0) \wedge \\
a(0) < b(0) < c(0) < a(1) &\wedge \\
\text{Adder3}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). &
\end{aligned}$$

Adder3 also behaves very much like *Adder1*, except that it always sequentially consumes an element from α first, then it consumes an element from β , then it produces their sum, before reading another element from α .

$$\begin{aligned}
\text{Adder4}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\
\gamma(0) &= \alpha(0) + \beta(0) \wedge \\
a(0) = b(0) = c(0) &\wedge \\
\text{Adder4}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). &
\end{aligned}$$

Adder4 behaves very much like *Adder1*, except that the consuming of every pair of input values and the production of their sum is one single atomic (synchronous) action.

$$\begin{aligned}
\text{Adder5}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) &\equiv \\
\gamma(0) &= \alpha(0) + \beta(0) \wedge \\
c(0) &= \min(a(1), b(1)) \wedge \\
\text{Adder5}(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle). &
\end{aligned}$$

Adder5 behaves very much like *Adder1*, except that it produces the sum of every pair atomically together with its reading of the first of its next pair of input values.

These examples show how the diluted notion of local time and its explicit representation in timed data streams enable us to concisely define and distinguish subtle differences in the behavior of various components that arise out of the delicate temporal order of their observable actions. The ability to make such distinctions differentiates otherwise equivalent behavior of similar components whose “equivalent behavior” leads to the Brock-Ackerman anomalies [13] concerning the input-output relation of components in nondeterministic dataflow models.

6.4 Analysis of ABT Compositions

Suppose we use *Adder4* of Section 6.3 to construct our Fibonacci circuit of Figure 8. Formally, the ABT models of the component *Adder4*, channels, and Reo nodes that we presented earlier suffice for an analysis of the behavior of their composition in this example. We briefly sketch such a formal analysis here to demonstrate the utility of the ABT model.

Let $\langle \alpha, a \rangle$ be the output of our system, as indicated in Figure 8. Form the ABT definition of the replicator (*Rpl*) inherent in the mixed node immediately on the left of this node, and the ABT definition of its three coincident **Sync** channels, we easily conclude that the output of *Adder4* and the input of **FIF01(1)** are also the same: $\langle \alpha, a \rangle$.

From the ABT definition of the **FIF01(1)** channel, we conclude that the sink end of this channel is the timed data stream $\langle 1.\alpha, b \rangle$, where $b < a < b'$. From the ABT definition of the replicator (*Rpl*) inherent in the mixed node at the output on this channel and the ABT definition of its coincident **Sync** channels, we conclude that the input to the **FIF01(0)** channel and the lower-input to *Adder4* are also the same timed data stream.

From the ABT definition of the **FIF01(0)** channel, we conclude that the output of this channel is the timed data stream $\langle 0.1.\alpha, c \rangle$, where $c < b < c'$. Given this as its input, the ABT definition of the **FIF01** channel yields $\langle 0.1.\alpha, d \rangle$ for its output, where $c < d < c'$.

The ABT definitions of the behavior of all of the above adders invariably yield

$\alpha = 0.1.\alpha + 1.\alpha$, which is simply a short-hand for the series of equations:

$$\begin{aligned}\alpha(0) &= 0 + 1 = 1 \\ \alpha(1) &= 1 + \alpha(0) = 1 + 1 = 2 \\ \alpha(2) &= \alpha(0) + \alpha(1) = 1 + 2 = 3 \\ \alpha(3) &= \alpha(1) + \alpha(2) = 2 + 3 = 5 \\ &\vdots\end{aligned}$$

Thus, α indeed represents the Fibonacci series.

However, the ABT definition of the behavior of *Adder4* requires $a = b = d$, whereas the condition on the output of the `FIFO1(1)` channel, above, states that $b < a < b'$. This leads to the contradiction of having both $a = b$ and $b < a$. What this contradiction tells us is that our composed system using *Adder4* will produce no output at all! ⁵

A closer examination reveals the reason: *Adder4* is a *synchronous* component; it must be able to consume both of its input values and produce its output, all in one single atomic step (i.e., transaction). The atomic reading of its lower input (b) together with the writing of its output (a) conflicts with the behavior of the `FIFO1(1)` channel. To comply with the behavior of *Adder4*, the `FIFO1(1)` channel must atomically both provide its output as the input to *Adder4*, and consume the output of *Adder4* as its own input. The ABT definition of the behavior of `FIFO1(1)` simply does not allow this to happen.

The only way to use such a synchronous adder as *Adder4* in this system, is to break this conflict, e.g., by replacing the `Sync` channel that connects the output of *Adder4* to the input of the `FIFO1(1)` channel, with a `FIFO1` channel.

On the other hand, our circuit in Figure 8 works perfectly if we use an adder with a different behavior, e.g., *Adder3*. The two adders produce the same data streams and the only difference between them is in their time streams. Using *Adder3*, we have $d < b < a < d'$. Because this equation implies $d < b$, which implies $d' < b'$, we can expand this equation as $d < b < a < d' < b'$, which complies with the $b < a < b'$ condition on the output of the `FIFO1(1)` channel, above. The timing conditions on the output of the `FIFO1(0)` channel ($c < b < c'$), and that of the `FIFO1` channel ($c < d < c'$) conform with the temporal constraints of *Adder3* as well. The assumption of dense time allows an infinity of viable solutions to the resulting system of equations. In the context of *Adder3*, what matters is that the `FIFO1` channel produces its output after it obtains the contents of the `FIFO1(0)` channel ($c < d$), but before the next input into the latter channel takes place ($c' < d'$ and $c' < b'$). Whether this next input occurs before *Adder3* writes its output ($c' < a$),

⁵ This example shows that the composition of two ABTs may yield the empty relation, which simply means the result has “no externally observable behavior.” Although “no externally observable behavior” can be interpreted as deadlock, there is nothing inherently wrong with or undesirable about it, because it can also be interpreted as normal termination. Thus, a composition that yields an empty ABT can be a perfectly legitimate way to model finite behavior in an otherwise perpetual systems. An example of such “desired deadlock” situations is presented in the inhibitor example in [5].

simultaneously ($c' = a$), or after ($a < c'$), is irrelevant.

Similarly, we can show that the behavior of *Adder1* or *Adder5* is also compatible with the context of the circuit in Figure 8 for producing the Fibonacci series. On the other hand, using *Adder2* in this circuit may or may not work. The behavior specification of *Adder2* allows it to always consume its *B* input (from the FIF01(1) channel) first. In this case, the circuit indeed produces the Fibonacci series. But, *Adder2* is also allowed to take its *A* input first. If *Adder2* always takes its *A* input first, then the circuit hangs and produces nothing at all, due to the same timing conflict as with *Adder4*. If *Adder2* internally decides afresh each time which input to take first, then the circuit will produce a finite sequence of the first $n \geq 0$ Fibonacci series, before it hangs and stops producing any further output.

Observe that all entities involved in this composed application are completely generic and, of course, neither knows anything about the Fibonacci series, nor the fact that it is “cooperating” with other entities to compute it. It is the specific glue code of this application, made by composing 8 simple generic channels in a specific topology in Reo, that coordinates the communication of the components (in this case, only one) with one another (in this case, with itself) and the environment to compute this series.

7 Petri Nets

Petri nets are frequently used to model interaction protocols and the behavior of complex systems. In some respects, Reo circuits resemble Petri nets. However, there are major differences between the two.

Petri nets are extensions of the finite state automata that incorporate a notion of concurrency. There are many different types of Petri nets, each of which extends the basic Petri net model with higher level concepts [27]. In this section, we consider only the elementary Petri nets, or the E/N systems. However, because we focus on the essential common features of all Petri nets, the distinctions we draw between Reo and the E/N systems also apply (with small alterations) to other Petri nets.

Petri nets consist of *places* and *transitions* with interconnecting *arcs*. Places can either be empty or hold *tokens*. In lower-level Petri nets, e.g., E/N systems, tokens are not distinguishable from one another. In colored Petri nets, each token can have a color that distinguishes it from the others. Multiple places can hold tokens in a Petri net at the same time. In E/N systems, each place can hold at most one token, but in higher-level Petri nets, a place can hold multiple tokens as well. The well-formedness condition of Petri nets ensures that an arc emanating from a place ends with a transition, and an arc emanating from a transition ends with a place. Multiple arcs can emanate and/or end at the same place or transition. In graphical models of Petri nets, transitions are often represented as solid rectangles; arcs as arrows; and places as either (1) hollow circles, if they are empty, or otherwise (2) circles that contain smaller (colored) solid circles representing their (colored) tokens. Figure 10 shows an example of a Petri net.

The *places*, *transitions* and *arcs* in Petri nets form a fixed set of building blocks,

each with a fixed behavior, for construction of Petri nets. In contrast, Reo defines a fixed set of composition rules and allows an arbitrary set of channels as primitives with arbitrary behavior, on which its composition rules can be applied to construct connector circuits. This readily allows incorporation of arbitrary computational entities into a composed Reo system. More importantly, it allows the harmonious combinations of synchrony and asynchrony in the same model which is not possible in Petri nets.

The similarity of the Petri net construction rules with Reo composition rules allows a direct translation of Petri nets into Reo circuits. Although direct translations of higher-level Petri nets into Reo circuits are also possible, here we consider only E/N systems.

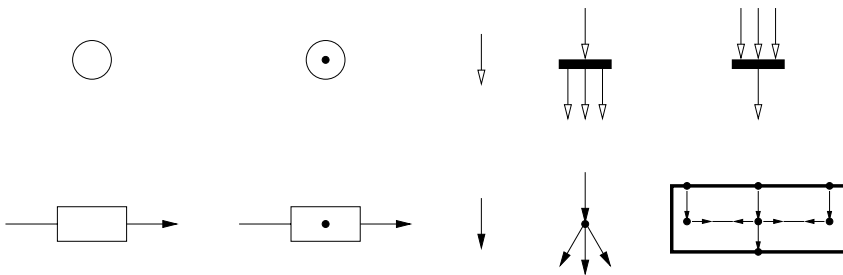


Fig. 9. Reo circuit equivalents for Petri net constructs

Figure 9 shows the Reo equivalent constructs (the bottom row) for Petri net building blocks (the top row). An empty place corresponds to a FIFO1 channel (see Figure 2 in Section 3.2.1). A filled place containing a token • corresponds to a FIFO1(•)⁶. An arc corresponds to a Sync channel. A transition with a single incoming arc and $n > 0$ outgoing arcs corresponds to a node with one incoming and n outgoing Sync channels. A transition with $m > 1$ incoming and $n > 0$ outgoing arcs corresponds to a degenerate barrier synchronizer (Figures 4.b and c in Section 4.2) Reo sub-circuit with $m - 1$ SyncDrain channels, m input nodes, and a single output node, as shown in the bottom-right of Figure 9. All n Sync channels that correspond to the outgoing arcs of this transition are connected to the single output node of this sub-circuit.

Using Figure 9, it is straight-forward to directly translate a Petri net into a Reo circuit. For example, applying this translation to the Petri net in Figure 10.a yields the Reo circuit in Figure 10.b. (The gray box in Figure 10.b represents a “degenerate barrier synchronizer” as shown in the lower-right corner of Figure 9.) In this sense, every Petri net can be trivially considered to be a Reo circuit. The inverse translation, however, is far from trivial.

In Reo, synchrony and exclusion constraints propagate through (the synchronous sub-sections of) circuits. This is generally not the case in Petri nets, because their transitions are local. What sets Petri nets apart from classical automata is their

⁶ In higher-level Petri nets a place can hold multiple tokens. Instead of (initialized or empty) FIFO1 channels, bag channels [5] must be used as their equivalents in Reo circuits (in the left two columns of the bottom row in Figure 9).

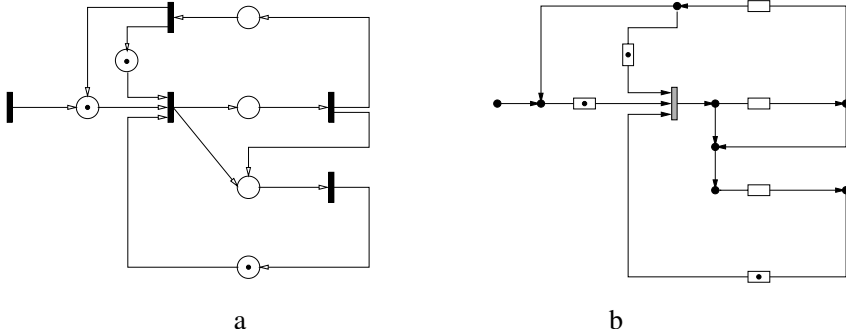


Fig. 10. Translation of Petri nets into Reo circuits

transition nodes, which enable them to directly synchronize otherwise unrelated events (it is no accident that a non-trivial Petri net transition node translates into a barrier synchronizer in Reo). A Petri net transition node enforces synchronous *and* of several arcs/events. However, Petri nets have no primitive for the dual synchronous *or* of several arcs, and there can be no arc between two places, nor between two transitions. The latter disallows nested *ands* of arcs. More significantly, the *or* of several arcs/transitions is possible only if they emanate from or end in the same place, which implies the commitment of moving a token from or into that place. This means that arcs/events can be directly *and*-synchronized to compose more complex synchronous transitions (i.e., one-step atomic transactions), but a synchronous *or* of arcs/events is not possible, i.e., two transitions cannot be connected together without an intervening place/commitment. This disallows a direct modeling of composite atomic transactions in Petri nets and prevents arbitrary combinations of synchrony and asynchrony.

The ability to construct arbitrarily complex synchronous sub-circuits (representing one-step atomic transactions) with asynchronous behavior in between, is unique in Reo and simplifies expressions of complex behavior. For example, it is non-trivial to construct the Petri net equivalents of the Reo circuits for barrier synchronization in Figures 4.b and c. In the context of e-commerce, [30] and [19] show the construction of non-trivial Reo circuits that implement negotiation protocols for competition and collaboration in electronic auctions. The Petri net models of these same protocols would be substantially more complex and elaborate, because they would have to “simulate” all atomic transactions involved.

8 Conclusion

The vast majority of classical models and paradigms for construction and study of complex systems use *actions* as their fundamental primitives. Examples include various object oriented programming models, the actor model [1], CSP [22], CCS [25], the π -calculus [26,29], and process algebras [10,11,21]. Because an action is something that a single actor performs, system construction in these models espouses a single-actor-at-a-time perspective. Complex global properties of a system involving more than one actor become obscure and difficult or impossible to verify and study,

because they cannot be expressed explicitly in these models.

Specification and study of global properties of complex systems become easier in a model that allows direct and explicit representation of interaction. Interaction can explicitly appear in the form of a relation that holds among a set of actors and constrains every one of them to coordinate their collective behavior. Such explicitly specified constraints can be composed together in various ways to yield more complex constraints (i.e., interaction protocols), without the need to specify the action sequences of any actors.

Reo is a good example of such a model. It offers (1) primitive interactions, in the form of channels, as building blocks, plus (2) composition rules for combining (primitive) interactions into more complex interactions (i.e., circuits), without the need to specify (the actions of) the actors involved. Indeed, every channel in Reo specifies a primitive interaction as a relational constraint that must hold between the I/O actions performed on its two ends, without saying anything about those actions or who performs them. These constraints specify the relative timing (i.e., synchrony/asynchrony) of (the success of) the I/O actions, and the desired data dependencies between them (e.g., buffering, ordering, selection, conversion, filtering, loss, and/or expiration of data). Reo's compositional operators compose such relations to produce the more complex constraints that constitute the behavior of their resulting connectors.

Our current and future work include development of various tools for (semi)-automatic reasoning, analysis, simulation, and animation of connector circuits, within a visual programming environment for Reo. Constraint automata and tools for their construction, composition, and model checking are an integral part of our on-going work.

References

- [1] Agha, G., "Actors: A Model of Concurrent Computation in Distributed Systems," MIT Press, 1986.
- [2] Aldrich, J., C. Chambers and D. Notkin, *Architectural reasoning in ArchJava*, in: B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Málaga, Spain, Proceedings*, Lecture Notes in Computer Science **2374** (2002), pp. 334–367.
- [3] Aldrich, J., C. Chambers and D. Notkin, *ArchJava: connecting software architecture to implementation*, in: *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)* (2002), pp. 187–197.
- [4] Arbab, F., *The IWIM model for coordination of concurrent activities*, in: P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, Lecture Notes in Computer Science **1061** (1996), pp. 34–56.
- [5] Arbab, F., *Reo: A channel-based coordination model for component composition*, *Mathematical Structures in Computer Science* **14** (2004), pp. 329–366.
- [6] Arbab, F., *Abstract Behavior Types: A foundation model for components and their composition*, *Science of Computer Programming* **55** (2005), pp. 3–52, extended version.
- [7] Arbab, F., C. Baier, J. Rutten and M. Sirjani, *Modeling component connectors in Reo by Constraint Automata*, in: *Proc. International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003)*, *Electronic Notes in Theoretical Computer Science (ENTCS)* **97** (2004), pp. 25–46, extended version to appear in the *Science of Computer Programming* journal, Elsevier, in 2006.
URL <http://www.elsevier.nl/locate/entcs>

- [8] Arbab, F. and J. Rutten, *A coinductive calculus of component connectors*, in: D. P. M. Wirsing and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT 2002)*, Lecture Notes in Computer Science **2755** (2003), pp. 35–56.
URL <http://www.cwi.nl/ftp/CWIREports/SEN/SEN-R0216.pdf>
- [9] Barringer, H., R. Kuiper and A. Pnueli, *A really abstract current model and its temporal logic*, in: *Proceedings of Thirteenth Annual ACM Symposium on principles of Programming Languages*, ACM, 1986, pp. 173–183.
- [10] Bergstra, J. A. and J. W. Klop, *Process algebra for synchronous communication*, Information and Control **60** (1984), pp. 109–137.
- [11] Bergstra, J. A. and J. W. Klop, *Process algebra: specification and verification in bisimulation semantics*, in: M. Hazewinkel, J. K. Lenstra and L. G. L. T. Meertens, editors, *Mathematics and Computer Science II*, CWI Monograph 4, North-Holland, Amsterdam, 1986 pp. 61–94.
- [12] Bonsangue, M., F. Arbab, J. de Bakker, J. Rutten, A. Scutellá and G. Zavattaro, *A transition system semantics for the control-driven coordination language Manifold*, Theoretical Computer Science **240** (2000), pp. 3–47.
- [13] Brock, J. and W. Ackerman, *Scenarios: A model of non-determinate computation*, in: *Proceedings of the International Colloquium on Formalization of Programming Concepts* (1981), pp. 252–259.
- [14] Broy, M. and G. Stefanescu, *The algebra of stream processing functions*, Theoretical Computer Science **258** (2001).
- [15] Broy, M. and K. Stolen, “Specification and development of interactive systems,” Monographs in Computer Science **62**, Springer, 2001.
- [16] *COM+*, <http://www.microsoft.com/com/tech/COMPlus.asp>.
URL <http://www.microsoft.com/com/tech/COMPlus.asp>
- [17] *CORBA*, <http://www.omg.org>.
URL <http://www.omg.org>
- [18] de Bakker, J. and J. Kok, *Towards a Uniform Topological Treatment of Streams and Functions on Streams*, in: W. Brauer, editor, *Proceedings of the 12th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **194** (1985), pp. 140–148.
- [19] Diakov, N., Z. Zlatev and S. Pokraev, *Composition of negotiation protocols for e-commerce applications*, in: W. Cheung and J. Hsu, editors, *The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, 2005, pp. 418–423.
- [20] *Enterprise JavaBeans*, <http://java.sun.com/products/ejb>.
URL <http://java.sun.com/products/ejb>
- [21] Fokkink, W., “Introduction to Process Algebra,” Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, 1999.
- [22] Hoare, C., “Communicating Sequential Processes,” Prentice Hall International Series in Computer Science, Prentice-Hall, 1985.
- [23] Kahn, G., *The semantics of a simple language for parallel programming*, in: J. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, North-Holland, New York, NY, 1974 pp. 471–475.
- [24] Kok, J., “Semantic Models for Parallel Computation in Data Flow, Logic- and Object-Oriented Programming,” Ph.D. thesis, Vrije Universiteit, Amsterdam (1989).
- [25] Milner, R., “A Calculus of Communicating Systems,” Lecture Notes in Computer Science **92**, Springer, 1980.
- [26] Milner, R., *Elements of interaction*, Communications of the ACM **36** (1993), pp. 78–89.
- [27] *Petri Nets World*, <http://www.informatik.uni-hamburg.de/TGI/PetriNets/>.
URL <http://www.informatik.uni-hamburg.de/TGI/PetriNets/>
- [28] Rutten, J., *Elements of stream calculus (an extensive exercise in coinduction)*, in: S. Brookes and M. Mislove, editors, *Proc. of 17th Conf. on Mathematical Foundations of Programming Semantics, Aarhus, Denmark, 23–26 May 2001*, Electronic Notes in Theoretical Computer Science **45**, Elsevier, Amsterdam, 2001 .
- [29] Sangiorgi, D. and D. Walker, “The Pi-Calculus - A Theory of Mobile Processes,” Cambridge University Press, 2001.
- [30] Zlatev, Z., N. Diakov and S. Pokraev, *Construction of negotiation protocols for E-Commerce applications*, ACM SIGecom Exchanges **5** (2004), pp. 11–22.