Puff, The Magic Protocol

Farhad Arbab[1,2]

[1] Foundations of Software Engineering, CWI, Science Park 123, 1098 XG Amsterdam
[2] Leiden Institute for Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
`farhad@cwi.nl`

Abstract. Traditional models of concurrency resort to peculiarly indirect means to express interaction and study its properties. Formalisms such as process algebras/calculi, concurrent objects, actors, agents, shared memory, message passing, etc., all are primarily action-based models that provide constructs for the direct specification of *things that interact*, rather than a direct specification of *interaction* (protocols). Consequently, interaction in these formalisms becomes a derived or secondary concept whose properties can be studied only indirectly, as the side-effects of the (intended or coincidental) couplings or clashes of the *actions* whose compositions comprise a model.

Treating interaction as an explicit first-class concept, complete with its own composition operators, allows to specify more complex interaction protocols by combining simpler, and eventually primitive, protocols. Reo [20,11,12,6] serves as a premier example of such an interaction-based model of concurrency. In this paper, we describe Reo and its support tools. We show how exogenous coordination in Reo reflects an interaction-centric model of concurrency where an interaction (protocol) consists of nothing but a relational constraint on communication actions. In this setting, interaction protocols become explicit, concrete, tangible (software) constructs that can be specified, verified, composed, and reused, independently of the actors that they may engage in disparate applications.

Puff, the magic dragon ad-libbed concurrency,
As he frolicked through the mist of code disguised invisibly.
Little Jackie Paper loved that rascal Puff,
And brought him threads and semaphores and other fancy stuff. Oh ...

$\neg PeterPaulAndMary$

1 Introduction

Concurrency is inherently difficult because it involves complex interaction protocols. Yet, it is always possible to make already difficult subjects even more difficult by increasing the complexity of their treatment. We take full advantage of this fact in cryptography by seeking easy disguising transformations whose inverses are so complex as to make them prohibitively difficult (if not impossible) to perform, without knowing a key (piece of information). We use unnecessary complexity to disguise things for entertainment, as in puzzles, for instance, as well. In most other situations, though, we do not choose to increase the complexity of a problem; at least not intentionally. However, it seems to me that the historically justifiable optimum path that led us out of the realm of sequential programming, into the new world of concurrency, has hindered us from realizing how our old-country world view unnecessarily increases the complexity of life in this new world. Once we had mastered the skills to navigate through the realm of sequential programming, the simplest models of concurrency seemed to require only the addition of a few new constructs to our models of sequential programming: a befitting selection of locks, semaphores, mutual exclusion, monitors, send/receive primitives, message passing, rendezvous, etc., would do. Refinements in concurrency theory abstracted away inconsequential sequential computations to offer process calculi and algebras. In many ways, these models are indeed simple. Alas, simple models are not always simple to use.

With the availability of today's low-cost multicore commodity hardware that can scale up to offer massively parallel computing platforms, high-speed communication networks that interconnect the globe, plus every indication that both of these phenomena constitute trends that will continue in the future, the need for programming techniques to harness the massive concurrency that they offer has become more vivid than ever. The inadequacy of traditional models for programming of concurrent systems to serve this purpose stems from the fact that the way in which they express interaction protocols generally does not scale up.

In spite of the fact that *interaction* constitutes the most challenging aspect of concurrency, traditional models of concurrency predominantly treat interaction as a secondary or derived concept. Shared memory, message passing, calculi such as CSP [50], CCS [83], the π-calculus [84,97], process algebras [33,25,46], and the actor model [8] represent popular approaches to tackle the complexities of constructing concurrent systems. Beneath their significant differences, all these models share one common characteristic, inherited from the world of sequential programming: they all constitute *action*-based models of concurrency.

For example, consider developing a simple concurrent application with two producers, which we designate as Green and Red, and one consumer. The consumer must repeatedly obtain and display the contents alternately made available by the Green and the Red producers.

Figure 1 shows the pseudo code for an implementation of this simple application in a Java-like language. Lines 1-4 in this code declare four globally shared entities: three semaphores and a buffer. The semaphores `greenSemaphore` and `redSemaphore` are used by their respective Green and Red producers for their turn keeping. The semaphore `bufferSemaphore` is used as a mutual exclusion lock for the producers and the consumer to access the shared `buffer`, which is initialized to contain the empty string. The rest of the code defines three processes: two producers and a consumer.

Global Objects:
```
 1 private final Semaphore greenSemaphore = new Semaphore(1);
 2 private final Semaphore redSemaphore = new Semaphore(0);
 3 private final Semaphore bufferSemaphore = new Semaphore(1);
 4 private String buffer = EMPTY;
```

Consumer:
```
 5 while (true) {
 6   sleep(4000);
 7   bufferSemaphore.acquire();
 8   if (buffer != EMPTY) {
 9     println(buffer);
10     buffer = EMPTY;
11   }
12   bufferSemaphore.release();
13 }
```

Green Producer:
```
14 while (true) {
15   sleep(5000);
16   greenText = ...;
17   greenSemaphore.acquire();
18   bufferSemaphore.acquire();
19   buffer = greenText;
20   bufferSemaphore.release();
21   redSemaphore.release();
22 }
```

Red Producer:
```
23 while (true)
24   sleep(3000);
25   redText = ...;
26   redSemaphore.acquire();
27   bufferSemaphore.acquire();
28   buffer = redText;
29   bufferSemaphore.release();
30   greenSemaphore.release();
31 }
```

Fig. 1. Alternating producers and consumer

The consumer code (lines 5-13) consists of an infinite loop where in each iteration, it performs some computation (which we abstract as the `sleep` on line 6), then it waits to acquire exclusive access to the buffer (line 7). While it has this exclusive access (lines 8-11), it checks to see if the buffer is empty. An empty buffer means there is no (new) content for the consumer process to display, in which case the consumer does nothing and releases the buffer lock (line 12). If the buffer is non-empty, the consumer prints its content and resets the buffer to empty (lines 9-10).

The Green producer code (lines 14-22) consists of an infinite loop where in each iteration, it performs some computation and assigns the value it wishes to produce to local variable `greenText` (lines 14-15), and waits for its turn by attempting to acquire `greenSsemaphore` (line 17). Next, it waits to gain exclusive access to the shared buffer, and while it has this exclusive access, it assigns `greenText` into `buffer` (lines 18-20). Having completed its turn, the Green producer now releases `redSemaphore` to allow the Red producer to have its turn (line 21).

The Red producer code (lines 23-31) is analogous to that of the Green producer, with "red" and "green" swapped.

This is a simple concurrent application whose code has been made even simpler by abstracting away its computation and declarations. Apart from their trivial outer infinite loops, each process consists of a short piece of sequential code, with a straight-line control flow that involves no inner loops or non-trivial branching. The protocol embodied in this application, as described in our problem statement, above, is also quite simple. One expects it be easy, then, to answer a number of questions about what specific parts of this code manifest the various properties of our application. For instance, consider the following questions:

1. Where is the green text computed?
2. Where is the red text computed?
3. Where is the text printed?

The answers to these questions are indeed simple and concrete: lines 16, 25, and 9, respectively. Indeed, the "computation" aspect of an application typically correspond to coherently identifiable passages of code. However, the perfectly legitimate question "Where is the protocol of this application?" does not have such an easy answer: the protocol of this application is intertwined with its computation code. More refined questions about specific aspects of the protocol have more concrete answers:

1. What determines which producer goes first?
2. What ensures that the producers alternate?
3. What provides protection for the global shared buffer?

The answer to the first question, above, is the collective semantics behind lines 1, 2, 17, and 26. The answer to the second question is the collective semantics behind lines 1, 2, 17, 26, 21, and 30. The answer to the third question is the collective semantics of lines 3, 18, 20, 27, and 29. These questions can be answered by pointing to fragments of code scattered among and intertwined with the computation of several processes in the application. It is far more difficult to identify other aspects of the protocol, such as possibilities for deadlock or live-lock, with concrete code fragments. While both concurrency-coordinating actions and computation actions are concrete and explicit in this code, the interaction protocol that they induce is implicit, nebulous, and intangible. In applications involving processes with even slightly less trivial control flow, the entanglement of data and control flow with concurrency-coordination actions makes it difficult to determine which parts of the code give rise to even the simplest aspects of their interaction protocol.

When the protocol in a typical concurrent application consists of 623 send and receive (or lock/unlock, etc.) primitives, sprinkled over 783,961 lines of C code, chopped up into 387 different source files, how simple is it to understand this protocol, reason about its properties, debug it, adapt it, or imagine reusing it in another application? How can a hapless programmer (who may very well be the original author of the code, six months down the road) even *see* what this protocol actually does before he can contemplate to do anything with it? Even in the case of our simple program in Figure 1, which we just examined, do we see all of its properties? We asked about and identified the buffer protection mechanism in this application. But does this mechanism provide adequate protection that we expect?

It is only tactful of me to say that I am sure all my readers have already spotted what may be considered a bug in this code that may in fact remain undetected in practice for a very long time, depending on the circumstances that determine the relative speeds of the producer and consumer threads in this application. There is no protection in this code preventing the producers from over-writing each other in the buffer, regardless of whether or not their output

Green Producer:

```
14 while (true) {
15   sleep(5000);
16   greenText = ...;
17   greenSemaphore.acquire();
18   while (greenText !=EMPTY) {
19     bufferSemaphore.acquire();
20     if (buffer == EMPTY) {
21       buffer = greenText;
22       greenText = EMPTY;
23     }
24     bufferSemaphore.release();
25   }
26   redSemaphore.release();
27 }
```

Red Producer:

```
28 while (true)
29   sleep(3000);
30   redText = ...;
31   redSemaphore.acquire();
32   while (redText !=EMPTY) {
33     bufferSemaphore.acquire();
34     if (buffer == EMPTY) {
35       buffer = redText;
36       redText = EMPTY;
37     }
38     bufferSemaphore.release();
39   }
40   greenSemaphore.release();
41 }
```

Fig. 2. Busy waiting consumer

has actually been consumed by the consumer. Strictly speaking, the original statement of our requirements does not forbid this behavior, so whether this is a bug (in the specification or implementation) is unclear. Suppose the intention in fact was for the consumer to alternately consume what the two producers produce, which means the implementation in Figure 1 is incorrect and we need to alter it.

One solution is to make the producers sensitive to the emptiness of the buffer. The code for the new producers appears in Figure 2. A disadvantage of this code is that it more heavily uses the busy-waiting mechanism that already existed in the consumer code in Figure 1. A better alternative is to use a different protocol that explicitly respects the turn taking, as described below.

In the program shown in Figure 3, the consumer too has its own turn-taking semaphore, the new `blueSemaphore` (line 3), which is initialized to be locked, just as the `redSemaphore`, because initially, there is nothing for the consumer to do before any of the producers produces something. The initialization of the `bufferSemaphore` is also changed (line 4), making the buffer initially locked on behalf of the first producer. The consumer and the two producers all can proceed until each reaches its own turn-taking lock on lines 8, 15, and 24, respectively. The consumer and the Red producer suspend themselves on their turn-taking locks, but the Green producer can proceed beyond its turn-taking lock (line 15), where it fills the buffer (line 16), releases the turn-taking lock of the consumer (line 17), and suspends itself on the buffer lock (line 18). Only the consumer can now proceed, printing the content of the buffer (line 9), and releasing the buffer lock (line 10), after which it proceeds with its next iteration in which it suspends itself on its turn-taking lock (line 8). Only the Green producer can now proceed, having obtained the buffer lock. It now completes its iteration by releasing the turn-taking lock of the Red producer (line 19), and starts its next iteration in which it suspends itself on its own turn-taking lock (line 15). Now, only the Red

Global Objects:
```
 1 private final Semaphore greenSemaphore = new Semaphore(1);
 2 private final Semaphore redSemaphore = new Semaphore(0);
 3 private final Semaphore blueSemaphore = new Semaphore(0);
 4 private final Semaphore bufferSemaphore = new Semaphore(0);
 5 private String buffer = EMPTY;
```

Consumer:
```
 6 while (true) {
 7   sleep(4000);
 8   blueSemaphore.acquire();
 9   println(buffer);
10   bufferSemaphore.release();
11 }
```

Green Producer:
```
12 while (true) {
13   sleep(5000);
14   greenText = ...;
15   greenSemaphore.acquire();
16   buffer = greenText;
17   blueSemaphore.release();
18   bufferSemaphore.acquire();
19   redSemaphore.release();
20 }
```

Red Producer:
```
21 while (true)
22   sleep(3000);
23   redText = ...;
24   redSemaphore.acquire();
25   buffer = redText;
26   blueSemaphore.release();
27   bufferSemaphore.acquire();
28   greenSemaphore.release();
29 }
```

Fig. 3. Revised alternating producers and consumer

producer can proceed to fill the buffer (line 25), release the turn-taking lock of the consumer (line 26), and suspend itself on the buffer lock (line 27). The consumer now goes through another iteration, at the end of which it releases the buffer lock, allowing only the Red producer to proceed. The Red producer now releases the turn-taking lock of the Green producer (line 29), and starts its next iteration in which it suspends itself on its own turn-taking lock (line 24) again.

Now that we have a correct protocol (if we indeed do) that does what we expect it to do (if it indeed does), what can we do with this protocol? How easy is it, for instance to reuse this same protocol in a more elaborate application where the control flow of the processes is more complex than the essentially linear, sequential flow of these simple processes? Is it possible to bundle up this protocol and parameterize it such that we can instantiate the protocol with arbitrary numbers of and computation code for processes, the same way that we can package a piece of code into a parameterized function to compute the inverse of a matrix of any size, or find the minimum element in a list of any size? It would certainly help in software development for multicore platforms, for instance, if we could simply specify the desired numbers and participants for an abstract parameterized protocol, as easily as passing arguments in a function call, to tailor the desired concurrency on the available cores. How easy is it to alter this protocol to change the imposed ordering or to allow a pair of considerably fast producers go as fast as they wish, while the slower consumer merely *samples* their output? Such manipulations are difficult with this and similar incarnations of a protocol because they require *seeing* and *touching* the protocol as a tangible concrete entity.

Seeing concurrency protocols through the mist of source code, reminds me of my experience with autostereograms that suddenly burst into popularity in the 1990's in *Magic Eye* books. In fact, there are different types of autostereograms and this particular type is called *random dot autostereograms* which hide a 3D image behind a pattern of seemingly random dots. The hidden 3D image emerges and becomes perceptible only when the incoherent 2D picture of random dots is viewed just the *right way*. To accomplish this feat, one needs to learn the skill to overcome the brain's normally automatic coordination between its mechanisms for the eyes' *focus* and *vergence*. With the correct vergence, the 3D scene suddenly pops into existence, but let the normal brain mechanism that ties vergence to focus take over, and puff, it's gone! It is inaccurate to call this phenomenon an optical illusion, because the 3D image is really there: all the depth information as well as its other characteristics truly exist embedded within the mist of random dots. It is nontrivial to learn the skill to see these 3D images because to do so is contrary to how our brains are wired to tell each eye where to look as we focus on what we see.

The protocol in a concurrent program is as real as the 3D image in a random dot autostereogram: all information necessary for its manifestation really exists, scattered, embedded within the bulk of the source code, most of which is just as irrelevant to the protocol and hinders its recognition as the random dots are to the 3D image and hinder its recognition. Seeing the protocol requires nontrivial skills that defy our natural balance of mental vergence and focus of attention. Constructing a random dot autostereogram requires intricate mathematical models and sophisticated calculations that do not resemble anything like sculpting or drawing a 3D image. Constructing a protocol in this form requires intricate mathematical models and sophisticated calculations that do not resemble anything like sequential programming. The 2D picture of a random dot autostereogram only indirectly contains its embedded 3D image, whose manifestation requires the active participation of an observer. The source code of a concurrent program only indirectly contains its embedded protocol, whose manifestation requires the active participation of a human or computer *observer*. Both the 3D images of random dot autostereograms and the protocols of concurrent programs can be constructed and manipulated only indirectly, through generally non-intuitive manipulations of seemingly unrelated tangible concrete objects scattered throughout the scene. Even the simplest manipulations of an autostereogram, such as scaling, can change the 3D image non-intuitively and produce strange unexpected results. It is just as perilous and misguided to try to alter a protocol or reuse (perhaps a part of) it in another program by directly manipulating or copying source code, as it is to try to alter a 3D image or reuse (perhaps a part of) it in another autostereogram by directly manipulating or copying random dots.

Process algebraic models of concurrency fair only slightly better in this regard than, e.g., programming with threads: they too embody an action-based model of concurrency. Figure 4 shows a process algebraic model of our alternating producers and consumer application. This model consists of a number of globally shared names, i.e., `g`, `r`, `b`, and `d`. Generally, these shared names are considered as abstractions of channels and thus are called "channels" in the process algebra/calculi community. However, since these names in fact serve no purpose

```
Global Names:                                   Green Producer:
  synchronization-points g, r, b, d               G := genG(t) . ?g(k) . !b(t) . ?d(j) . !r(k) . G

Consumer:                                       Red Producer:
  B :=  ?b(t) . print(t) . !d("done") . B        R := genR(t) . ?r(k) . !b(t) . ?d(j) . !g(k) . R

Application:
  G | R | B | !g("token")
```

Fig. 4. Alternating producers and consumer in a process algebra

other than synchronizing the I/O operations performed on them, and because we will later use the term "channel" to refer to entities with more elaborate behavior, we use the term "synchronization points" here to refer to "process algebra channels" to avoid confusion.

A process algebra consists of a set of atomic actions, and a set of composition operators on these actions. In our case, the atomic actions include the primitive actions read `?_(_)` and write `!_(_)` defined by the algebra, plus the user-defined actions `genG(_)`, `genR(_)`, and `print(_)`, which abstract away computation. Typical composition operators include sequential composition `_ . _`, parallel composition `_ | _`, nondeterministic choice `_ + _`, definition `_ := _`, and implicit recursion.

In our model, the consumer `B` waits to read a data item into `t` by synchronizing on the global name `b`, and then proceeds to print `t` (to display it). It then writes a token `"done"` on the synchronization point `d`, and recurses. The Green producer `G` first generates a new value in `t`, then waits for its turn by reading a token value into `k` from `g`. It then writes `t` to `b`, and waits to obtain an acknowledgment `j` through `d`, after which it writes the token `k` to `r`, and recurses. The Red producer `R` behaves similarly, with the roles of `r` and `g` swapped. The application consists of a parallel composition of the two producers and the consumer, plus a trivial process that simply writes a `"token"` on `g` to kick off process `G` to go first.

Observe that a model is constructed by composing (atomic) actions into (more complex) actions, called processes. True to their moniker, such formalisms are indeed *algebras of processes* or actions. Just as in the version in Figure 3, while communication actions are concrete and explicit in the incarnation of our application in Figure 4, *interaction* is a manifestation of the model with no direct explicit structural correspondence. Nevertheless, process algebraic incarnations of concurrency protocols are obviously simpler and more concise than their incarnations in typical programming languages, primarily because they abstract away the clutter of computation.

Returning to our autostereogram analogy, it is as if we compare a random dot autostereogram with a so-called *wallpaper autostereogram*. A wallpaper autostereogram is the simplest type of autostereogram and consists of a horizontally repeating pattern of nearly identical pictures. Roughly, it is the random dot autostereogram with the cluttering random dots peeled off, which allows even casual observers to get a good idea of what the 3D image is all about, without requiring them to exert their perception skills to actually experience the 3D image. The individually identifiable repeating patterns of a wallpaper autostereogram seem more concrete, and in some sense more well-packaged and more reusable than the almost amorphous expanse of a random dot autostereogram. Nevertheless, a cavalier attempt to edit or cut and paste parts of a wallpaper autostereogram is no more likely to produce the desired alteration to its 3D image than in the case of a random dot autostereogram. Successful alteration of a process algebraic specification requires the same unnatural detachment of focus (on local manipulation) and vergence (to see its global effects) as is required to successfully alter the protocol of a concurrent C or Java application.

Indeed, in all action-based models of concurrency, interaction becomes a by-product of processes executing their respective actions: when a process A happens to execute its i_{th} communication action a_i on a synchronization point, at the same time that another process B happens to execute its j_{th} communication action b_j on that same synchronization point, the actions a_i and b_j "collide" with one another and their collision yields an interaction. Manifested this way, an interaction protocol consists of a desired temporal sequence of such (coincidental or planned) collisions. It is non-trivial to distinguish between the essential and the coincidental collision sequences, when the protocol itself is only such an ephemeral manifestation.

Generally, the reason behind the specific collision of a_i and b_j remains debatable. Perhaps it was just dumb luck. Perhaps it was divine intervention. Some may prefer to attribute it to intelligent design! What is not debatable is the fact that, a split second earlier or later, perhaps in another run of the same application, completely random cosmic rays may zap a memory bit and trigger the automatic hardware error correction of the affected memory cell, and thus change the relative timing of the running processes, making a_i and b_j collide not with each other, but with two other actions (of perhaps other processes) yielding completely different interactions. Action based models of concurrency make protocols more difficult than necessary to specify, manipulate, verify, debug, and next to impossible to reuse.

Instead of explicitly composing (communication) actions to indirectly specify and manipulate implicit interactions, is it possible to devise a model of concurrency where interaction (not action) is an explicit, first-class construct? We tend to this question in the next section and in the remainder of this paper describe a specific language based on an interaction-centric model of concurrency. We show that making interaction explicit leads to a clean separation of computation and communication, and produces reusable, tangible protocols that can be constructed and verified independently of the processes that they engage.

2 Interaction Centric Concurrency

The most salient characteristic of *interaction* is that it transpires among two or more actors. This is in contrast to *action*, which is what a single actor manifests. In other words, interaction is not about the specific actions of individual actors, but about the relations that (must) hold among those actions. A model of interaction, thus, must allow us to directly specify, represent, construct, compose, decompose, analyze, and reason about those relations that define what transpires among two or more engaged actors, without the necessity to be specific about their individual actions. Making interaction a first-class concept means that a model must offer (1) an explicit, direct, concrete representation of the interaction among actors, independent of their (communication) actions; (2) a set of primitive interactions; and (3) composition operators to combine (primitive) interactions into more complex interactions.

Wegner has proposed to consider coordination as constrained interaction [101]. We propose to go a step further and consider interaction itself as a constraint on (communication) actions. Features of a system that involve several entities, for instance the clearance between two physical objects, cannot conveniently be associated with any one of those entities. It is quite natural to specify and represent such features as *constraints*. The interaction among several active entities has a similar essence: although it involves them, it does not *belong* to any one of those active entities. Constraints have a natural formal model as mathematical relations, which are non-directional. In contrast, actions correspond to functions or mappings which are directional, i.e., transformational.

A constraint declaratively specifies *what* must hold in terms of a relation. Typically, there are many ways in which a constraint can be enforced or violated, leading to many different sequences of actions that describe precisely *how* to enforce or maintain a constraint. Action-based models of concurrency lead to the precise specification of *how* in terms of sequences of actions interspersed among the active entities involved in a protocol. In an interaction-based model of concurrency, only *what* a protocol represents is specified as a constraint over the (communication) actions of some active entities; as in constraint programming, the responsibility of how the protocol constraints are enforced or maintained is relegated to an entity other than those active entities.

Generally, composing the sequences of actions that manifest two different protocols does not yield a sequence of actions that manifests a composition of those protocols. Thus, in action-based models of concurrency, protocols are not compositional. Represented as constraints, in an interaction-based model of concurrency, protocols can be composed as mathematical relations.

Banishing the actions that comprise protocol fragments out of the bodies of processes produces simpler, cleaner, and more reusable processes. Expressed as constraints, pure protocols become first-class, tangible, reusable constructs in their own right. As concrete software constructs, such protocols can be embodied into architecturally meaningful *connectors*.

In this setting, a process (or thread, component, service, actor, agent, etc.) offers no methods, functions, or procedures for other entities to call, and it makes no such calls itself. Moreover, processes cannot exchange messages through targeted send and receive actions. In fact, a process cannot refer to any foreign entity, such as another process, the mailbox or message queue of another process, shared variables, semaphores, locks, etc. The only means of communication of a process with its outside world is through *blocking I/O operations* that it may perform exclusively on its own *ports*, producing and consuming passive data.

A port is a construct analogous to a file descriptor in a Unix process, except that a port is unidirectional, has no buffer, and supports blocking I/O exclusively.

If `i` is an input port of a process, there are only two operations that the process can perform on `i`: (1) blocking input `get(i, v)` waits indefinitely or until it succeeds to obtain a value through `i` and assigns it to variable `v`; and (2) input with time-out `get(i, v, t)` behaves similarly, except that it unblocks and returns false if the specified time-out `t` expires before it obtains a value to assign to `v`. Analogously, if `o` is an output port of a process, there are only two operations that the process can perform on `o`: (1) blocking output `put(o, v)` waits indefinitely or until it succeeds to dispense the value in variable `v` through `o`; and (2) output with time-out `put(o, v, t)` behaves similarly, except that it unblocks and returns false if the specified time-out `t` expires before it dispenses the value in `v`.

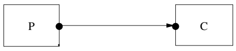

Fig. 5. Protocol in a connector

Inter-process communication is possible only by mediation of connectors. For instance, Figure 5 shows a producer, `P` and a consumer `C` whose communication is coordinated by a simple connector. The producer `P` consists of an infinite loop in each iteration of which it computes a new value and writes it to its local output port (shown as a small circle on the boundary of its box in the figure) by performing a blocking `put` operation. Analogously, the consumer `C` consists of an infinite loop in each iteration of which it performs a blocking `get` operation on its own local input port, and then uses the obtained value. Observe that, written in an imperative programming language, the code for `P` and `C` is substantially simpler than the code for the Green/Red producers and the consumer in Figures 1, 2, and 3: it contains no semaphore operations or any other inter-process communication primitives.

The direction of the connector arrow in Figure 5 suggests the direction of the dataflow from `P` to `C`. However, even in the case of this very simple example, the precise behavior of the system crucially depends on the specific protocol that this simple connector implements. For instance, if the connector implements a synchronous protocol, then it forces `P` and `C` to iterate in lock-step, by synchronizing their respective `put` and `get` operations in each iteration. On the other hand the connector may have a bounded or an unbounded buffer and implement an asynchronous protocol, allowing `P` to produce faster than `C` can consume. The protocol of the connector may, for instance enable it to replicate data items, e.g., the last value that it contained, if `C` consumes faster and drains the buffer. The protocol may mandate an ordering other than FIFO on the contents of the connector buffer, perhaps depending on the contents of the exchanged data. It may retain only some of the contents of the buffer (e.g., only the first or the last item) if P produces data faster than C can consume. It may be unreliable and lose data nondeterministically or according to some probability distribution. It may retain data in its buffer only for a specified length of time, losing all data items that are not consumed before their expiration dates. The alternatives for the connector protocol are endless, and composed with the very same P and C, each yields a totally different system.

A number of key observation about this simple example are worth noting. First, Figure 5 is an architecturally informative representation of this system. Second, banishing all inter-process communication out of the communicating parties, into the connector, yields a "good" system design with the beneficial consequences that:

– changing P, C, or the connector does not affect the other parts of the system;
– although they are engaged in a communication with each other, P and C are oblivious to each other, as well as to the actual protocol that enables their communication;
– the protocol embodied in the connector is oblivious to P and C.

In this architecture, the composition of the components and the coordination of their interactions are accomplished *exogenously*, i.e., from outside of the components themselves, and without their "knowledge"[1]. In contrast, the interaction protocol and coordination in the examples in Figures 1, 2, 3, and 4 are *endogenous*, i.e., accomplished through (inter-process communication) primitives from inside the parties engaged in the protocol. It is clear that exogenous composition and coordination lead to simpler, cleaner, and more reusable component code, simply because all composition and coordination concerns are left out. What is perhaps less obvious is that exogenous coordination also leads to reusable, pure coordination code: there is nothing in any incarnation of the connector in Figure 5 that is specific to P or C; it can just as readily engage any producer and consumer processes in any other application.

Obviously, we are not interested in only this example, nor exclusively in connectors that implement exogenous coordination between only two communicating parties. Moreover, the code for any version of the connector in Figure 5, or any other connector, can be written in any programming language: the concepts of exogenous composition, exogenous coordination, and the system design and architecture that they induce constitute what matters, not the implementation language. Focusing on multi-party interaction/coordination protocols reveals that they are composed out of a small set of common recurring concepts. They include synchrony, atomicity, asynchrony, ordering, exclusion, grouping, selection, etc. Compliant with the constraint view of interaction advocated above, these concepts can be expressed as constraints, more directly and elegantly than as compositions of actions in a process algebra or an imperative programming

[1] By this anthropomorphic expression we simply mean that a component does not contain any piece of code that directly contributes to determine the entities that it composes with, or the specific protocol that coordinates its own interactions with them.

language. This observation behooves us to consider the interaction-as-constraint view of concurrency as a foundation for a special language to specify multi-party exogenous interaction/coordination protocols and the connectors that embody them, of which the connector in Figure 5 is but a trivial example. Reo, described in the next section, is a premier example of such a language.

3 An Overview of Reo

Reo [20,11,12,6] is a channel-based exogenous coordination language wherein complex coordinators, called connectors, are compositionally built out of simpler ones. Exogenous coordination imposes a purely local interpretation on each inter-components communication, engaged in as a pure I/O operation on each side, that allows components to communicate anonymously, through the exchange of untargeted passive data. We summarize only the main concepts in Reo here. Further details about Reo and its semantics can be found in the cited references.

Complex connectors in Reo are constructed as a network of primitive binary connectors, called *channels*. Connectors serve to provide the protocol that controls and organizes the communication, synchronization and cooperation among the components/services that they interconnect. Formally, the protocol embodied in a connector is a *relation*, which the connector imposes as a *constraint* on the actions of the communicating parties that it inter-connects.

A channel is a medium of communication that consists of two ends and a constraint on the dataflows observed at those ends. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. Every channel (type) specifies its own particular behavior as constraints on the flow of data through its ends. These constraints relate, for example, the content, the conditions for loss, and/or creation of data that pass through the ends of a channel, as well as the atomicity, exclusion, order, and/or timing of their passage. Reo places no restriction on the behavior of a channel and thus allows an open-ended set of different channel types to be used simultaneously together.

Although all channels used in Reo are user-defined and users can indeed define channels with any complex behavior (expressible in the semantic model) that they wish, a very small set of channels, each with very simple behavior, suffices to construct useful Reo connectors with significantly complex behavior. Figure 6 shows a common set of primitive channels often used to build Reo connectors.

A `Sync` channel has a source and a sink end and no buffer. It accepts a data item through its source end iff it can simultaneously (i.e., atomically) dispense it through its sink.

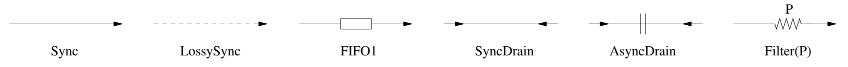

Fig. 6. A typical set of Reo channels

A `LossySync` channel is similar to a synchronous channel except that it always accepts all data items through its source end. This channel transfers a data item if it is possible for the channel to dispense the data item through its sink end; otherwise the channel loses the data item. Observe that the behavior of this channel if fully deterministic; the channel is never free to choose between passing or losing a data item: if it is possible for a data item to be consumed through its sink end, the channel *must* pass the data item exactly as a `Sync`. Thus, the context of (un)availability of a ready consumer at its sink end determines the (context-sensitive) behavior a `LossySync` channel.

A `FIFO1` channel represents an asynchronous channel with a buffer of capacity 1: it can contain at most one data item. In the graphical representation of an empty `FIFO1` channel, no data item is shown in the box (this is the case in Figure 1). If the buffer of a `FIFO1` channel contains a data element *d*, then *d* appears inside the box in its graphical representation. When its buffer is empty, a `FIFO1` channel blocks I/O operations on its sink, because it has no data to dispense. It dispenses a data item and allows an I/O operation at its sink to succeed, only when its buffer is full, after which its buffer becomes empty. When its buffer is full, a `FIFO1` channel blocks I/O operations on its source, because it has no more capacity to accept the incoming data. It accepts a data item and allows an I/O operation at its source to succeed, only when its buffer is empty, after which its buffer becomes full.

More exotic channels are also permitted in Reo, for instance, synchronous and asynchronous *drains*. Each of these channels has two source ends and no sink end. No data value can be obtained from a drain channel because it has no sink end. Consequently, all data accepted by a drain channel are lost. `SyncDrain` is a synchronous drain that can accept a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. `AsyncDrain` is an asynchronous drain that accepts data items through its source ends and loses them exclusively one at a time, but never simultaneously.

For a *filter channel*, or `Filter(P)`, its pattern $\mathtt{P} \subseteq \mathit{Data}$ specifies the type of data items that can be transmitted through the channel. This channel accepts a value $d \in \mathtt{P}$ through its source end iff it can simultaneously dispense *d* through its sink end, exactly as if it were a `Sync` channel; it always accepts all data items $d \notin \mathtt{P}$ through its source end and loses them immediately.

Synchronous and asynchronous *Spouts* are the duals of their respective drain channels, as each has two sink ends through which it produces nondeterministic data items. Further discussion of these and other primitive channels is beyond the scope of this paper.

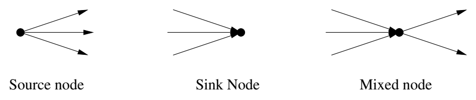

Fig. 7. Reo nodes

Complex connectors are constructed by composing simpler ones via the *join* and *hide* operations. Channels are joined together in *nodes*, each of which consists of a set of channel ends. A Reo node is a logical place where channel ends coincide and coordinate their dataflows as prescribed by its *node type*. Figure 7 shows the three possible node types in Reo. A node is either *source*, *sink*, or *mixed*, depending on whether all channel ends that coincide on that node are source ends, sink ends, or a combination of the two. Reo fixes the semantics of (i.e., the constraints on the dataflow through) Reo nodes, as described below. The *hide* operation is used to hide the internal topology of a Reo connector. A hidden nodes can no longer be accessed or observed from outside.

The source and sink nodes of a connector are collectively called its *boundary nodes*. Boundary nodes define the interface of a connector. Processes (or components, actors, agents, etc.) connect to the boundary nodes of a connector and interact anonymously with each other through this interface. Connecting a process to a (source or sink) node of a connector consists of the identification of one of the (respectively, output or input) ports of the component with that node. At most one process can be connected to a (source or sink) node at a time. Processes interact by performing their blocking I/O operations on their own local ports, which trigger dataflow through their respectively identified nodes of the connector(s): the `get` and `put` operations mentioned in the description of the components in Figure 5 trigger *write* and *take* operations of Reo on the channel ends of their respective nodes.

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 synchronous replicator.

A component can obtain data items, by an input operation, from a sink node that it is connected to. 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 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. Note that a component cannot connect to, take from, or write to mixed nodes.

Because a node has no buffer, data cannot be stored in a node. Specifically, a mixed node cannot take a data item out of one of its coincident sink channel ends, unless it can atomically replicate and write it into all of its coincident source channel ends. Hence, nodes instigate the propagation of synchrony and exclusion constraints on dataflow throughout a connector. Deriving the semantics of a Reo connector amounts to resolving the composition of the constraints of its constituent channels and nodes [43]. This is not a trivial task. In the sequel, we present examples of Reo connectors that illustrate how non-trivial dataflow behavior emerges from composing simple channels using Reo nodes. The local constraints of individual channels propagate through (the synchronous regions of) a connector to its boundary nodes. This propagation also induces a certain context-awareness in connectors. See [41] for a detailed discussion of this.

Reo has been used for composition of Web services [65,77,21], modeling and analysis of long-running transactions in service-oriented systems [69], coordination of multi-agent systems [13], performance analysis of coordinated compositions [23,16,17,86,87], modeling of business processes and verification of their compliance [98,68,19], and modeling of coordination in biological systems [40].

Reo offers a number of operations to reconfigure and change the topology of a connector at run-time: operations that enable the dynamic creation of channels, splitting and joining of nodes, hiding internal nodes. The hiding of internal nodes allows to permanently fix the topology of a connector, such that only its boundary nodes are visible and available. The resulting connector can then be viewed as a new primitive connector, or primitive for short, since its internal structure is hidden and its behavior is fixed.

4 Examples

Recall our alternating producers and consumer example of Section 1. We revise the code for the Green and Red producers to make them suitable for exogenous coordination (which, in fact, makes them simpler). Similar to the producer P in Figure 5, this code now consists of an infinite loop, in each iteration of which the producer computes a new value and writes it to its output port. Analogously, we revise the consumer code, fashioning it after the consumer C in Figure 5. Figure 8 shows this code.

Consumer:
```
while (true) {
  sleep(4000);
  get(input, displayText);
  print(displayText);
}
```

Green Producer:
```
while (true) {
  sleep(5000);
  greenText = ...;
  put(output, greenText);
}
```

Red Producer:
```
while (true)
  sleep(3000);
  redText = ...;
  put(output, redText);
}
```

Fig. 8. Generic reusable producers and consumer

In the remainder of this section, we present a number of protocols to implement different versions of the alternating producers and consumer example of Section 1, using the producers and consumer processes in Figure 8. These examples serve three purposes. First, they show a flavor of programming of pure interaction coordination protocols as Reo circuits. Second, they present a number of generically useful circuits that can serve as connectors in many other applications, or as sub-circuits in the circuits for construction of many other protocols. Third, they illustrate the utility of exogenous coordination by showing how trivial it is to change the protocol of an application, without altering any of the processes involved.

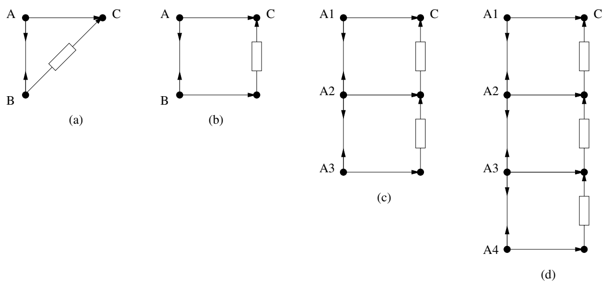

Fig. 9. Reo circuits for Alternators

4.1 Alternator

The connector shown in Figure 9(a) is an *alternator* that imposes an ordering on the flow of the data from its input nodes A and B to its output node C. The `SyncDrain` enforces that data flow through A and B only synchronously (i.e., atomically). The empty buffer of the the `FIFO1` channel together with the `SyncDrain` guarantee that the data item obtained from A is delivered to C while the data item obtained from B is stored in the `FIFO1` buffer. After this, the buffer of the `FIFO1` is full and data cannot flow in through either A or B, but C can dispense the data stored in the `FIFO1` buffer, which makes it empty again. Thus, subsequent take operations at C obtain the data items written to $A, B, A, B, ...,$ etc.

The connector in Figure 9(b) has an extra `Sync` channel between node B and the `FIFO1` channel, compared to the one in Figure 9(a). It is trivial to see that these two connectors have the exact same behavior. However, the structure of the connector in Figure 9(b) allows us to generalize its alternating behavior to any number of producers, simply by replicating it and "juxtaposing" the top and the bottom `Sync` channels of the resulting copies, as seen in Figure 9(c) and Figure 9(d).

The two `SyncDrain` channels in the connector shown in Figure 9(c) require data to flow through $A1$, $A2$, and $A3$ only simultaneously (i.e., atomically). The empty buffers of the `FIFO1` channels, together with these `SyncDrain` channels guarantee that the data item obtained from $A1$ is delivered to C while the data items obtained from $A2$ and $A3$ are stored in the buffers of their respective `FIFO1` channels. Subsequently, as long as the buffer of at least one of the `FIFO1` channels remains full, no data can flow through any of the nodes $A1$, $A2$, and $A3$, but C can dispense the data stored in the buffers of the `FIFO1` channels, with their order preserved. Thus, the first 3 take operations on C deliver the data items obtained through $A1$, $A2$, and $A3$, in that order. At this point, all `FIFO1` buffers become empty and the next round of input becomes possible.

The connector in Figure 9(d) is obtained by replicating the one in Figure 9(b) 3 times. Following the reasoning for the connector in Figure 9(c), it is easy to see that the connector in Figure 9(d) delivers the data items obtained from $A1$, $A2$, $A3$,and $A4$ through C, in that order.

A version of our alternating producers and consumer example of Section 1 can now be composed by attaching the output port of the revised Green producer in Figure 8 to node A, the output port of the revised Red producer in Figure 8 to node B, and the input port of the consumer in Figure 8 to node C of the Reo circuit in Figure 9(a).

A closer look shows, however, that the behavior of this version of our example is *not* exactly the same as that of the one in Figures 3 and 4. As explained above, the Reo circuit in Figure 9(a) requires the availability of a pair of values on A (from the Green producer) and B (from the Red producer) before it allows the consumer to obtain them, first from A and then from B. Thus, if the Green producer and the consumer are both ready to communicate, they still have to wait for the Red producer to also attempt to communicate, before they can exchange data. The versions in Figures 3 and 4 allow the Green producer and the consumer to go ahead, regardless of the state of the Red producer. Our original specification of this example in Section 1 was abstract enough to allow both alternatives. A further refinement of this specification may indeed prefer one and disallow the other. If the behavior of the connector in Figure 9(a) is *not* what we want, we need to construct a different Reo circuit to impose the same behavior as in Figures 3 and 4. This is precisely what we describe below.

4.2 Sequencer

Figure 10(a) shows an implementation of a sequencer by composing five `Sync` channels and four `FIFO1` channels together. The first (leftmost) `FIFO1` channel is initialized to have a data item in its buffer, as indicated by the presence of the symbol e in the box representing its buffer cell. The actual value of the data item is irrelevant. The connector provides only the four nodes A, B, C and D for other entities (connectors or component instances) to take from. The take operation on 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.

Figure 10(b) shows a simple example of the utility of the sequencer. The connector in this figure consists of a two-node sequencer, plus a `SyncDrain` and two `Sync` channels connecting each of the nodes of the sequencer to the nodes A and C, and B and C, respectively. Similar to the circuit in Figure 9(a), this connector imposes an order on the flow of the data items written to A and B, through C: the sequence of data items obtained by successive take operations

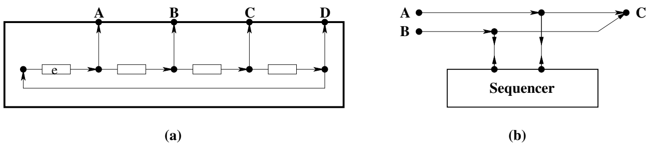

Fig. 10. Sequencer

on C consists 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, and so on. However, there is a subtle difference between the behavior of the two circuits in Figures 9(a) and 10(b). The alternator in Figure 9(a) delays the transfer of a data item from A to C until a data item is also available at B. The circuit in Figure 10(b) transfers from A to C as soon as these nodes can satisfy their respective operations, regardless of the availability of data on B.

We can obtain a new version of our alternating producers and consumer example by attaching the output port of the Green producer in Figure 8 to node A, the output port of the Red producer in Figure 8 to node B, and the input port of the consumer in Figure 8 to node C. The behavior of this version of our application is now the same as the programs in Figure 4 and in Figure 1 (after replacing its producers with the ones in Figure 2). The circuit in Figure 10(b) embodies the same protocol that is implicit in Figure 4.

A characteristic of this protocol is that it "slows down" each producer, as necessary, by delaying the success of its data production until the consumer is ready to accept its data. Our original problem statement in Section 1 does not explicitly specify whether or not this is a required or permissible behavior. While this may be desirable in some applications, slowing down the producers to match the processing speed of the consumer may have serious drawbacks in other applications, e.g., if these processes involve time-sensitive data or operations. Perhaps what we want is to bind our producers and consumer by a protocol that decouples them such as to allow each process to proceed at its own pace. We proceed, below, to present a number of protocols that we then compose to construct a Reo circuit for such a protocol.

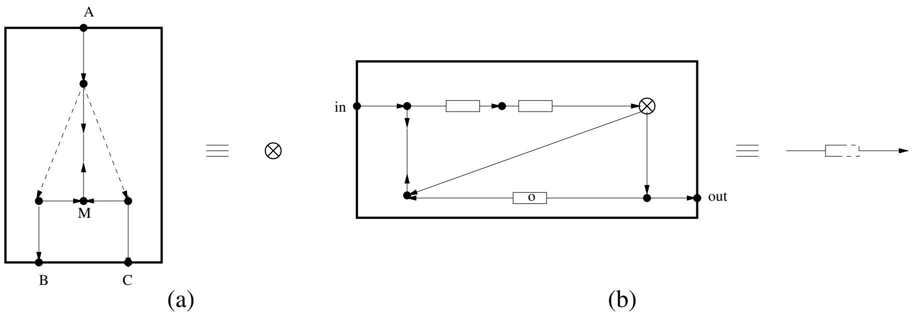

Fig. 11. An exclusive router and a `ShiftLossyFIFO1`

4.3 Exclusive Router

The connector shown in Figure 11(a) is a binary *exclusive router*: it routes data from A to either B or C (but not both). This connector can accept data only if there is a write operation at the source node A, and there is at least one taker at the sink node B or C. If both B and C can dispense data, the choice of routing to B or C follows from the non-deterministic decision by the mixed node M: it can accept data only from one of its sink ends, excluding the flow of data through the other, which forces the latter's respective LossySync to lose the data it obtains from A, while the other LossySync passes its data as if it were a Sync.

By connecting the source node of a binary exclusive router to one of the sink nodes of another binary exclusive router we obtain a ternary exclusive router. This is possible in Reo because synchrony and exclusion constraints propagate through its nodes. More generally, an n-ary exclusive router (with a single source and n sink ends) can be composed out of $n-1$ binary exclusive routers. Because the exclusive routers are so commonly useful, we use a graphical short-hand to represent them in circuits. The crossed circle shown on the right-hand side of Figure 11(a) is the symbol that we use to represent a generic n-ary exclusive router.

4.4 Shift-Lossy FIFO1

Figure 11(b) shows a Reo circuit for a connector that behaves as a lossy FIFO1 channel with a shift loss-policy. This channel is called shift-lossy FIFO1 (ShiftLossyFIFO1). This connector is composed of an exclusive router (shown in Figure 11(a)), an initially full FIFO1 channel, two initially empty FIFO1 channels, and four Sync channels. Intuitively, it behaves as a normal FIFO1 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 older contents in its buffer in favor of the newer arrivals. This is in contrast to the behavior of an overflow-lossy FIFO1 channel, whose "overflow loss-policy" loses the new arrivals when its buffer is full. See [31] for a more formal treatment of the semantics of this connector.

The ShiftLossyFIFO1 circuit in Figure 11(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. The symbol shown on the right-hand side of Figure 11(b) is the what we use to represent this circuit, and also take the liberty to refer to it as a ShiftLossyFIFO1 "channel". This symbol is intentionally similar to that of a regular FIFO1 channel, because the behavior of this circuit closely resembles that of a regular FIFO1 channel. The dashed sink-side half of the box representing the buffer of this channel suggests that it loses the older values to make room for new arrivals, i.e., it shifts to lose.

4.5 Decoupled Alternating Producers and Consumer

Figure 12(a) shows how the ShiftLossyFIFO1 circuit of Figure 11(b) can be used to construct a version of the example in Figure 5, where the producer and the consumer are partially decoupled from one another. Whenever, as initially is the case, the ShiftLossyFIFO1 buffer is empty, the consumer has no choice but to wait for the producer to place a value into this buffer. However, the producer never has to wait for the consumer: it can work at its own pace and write to the connector whenever it wishes. Every write by the producer replaces the current contents of the `ShiftLossyFIFO1` buffer. A subsequent take by the consumer obtains the current value out of `ShiftLossyFIFO1` buffer and makes it empty. The producer never has to wait for the consumer, but if the consumer is faster than the producer, it has to wait for the next data item to arrive. It is instructive to compare the behavior of this system with that of a single `LossySync` channel connecting a producer and a consumer: the two are not exactly the same.

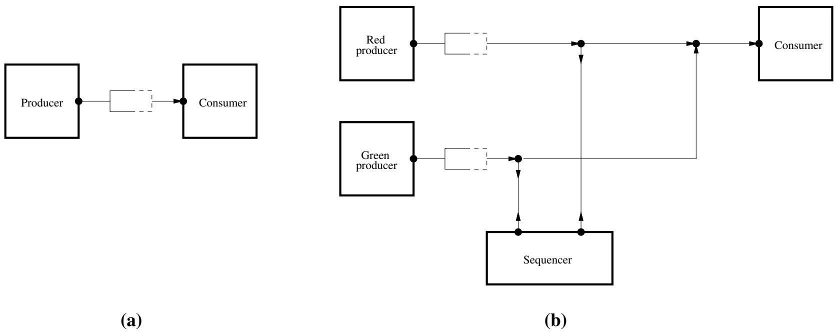

Fig. 12. Decoupled producers and consumer

The connector in Figure 12(b) is a small variation of the Reo circuit in Figure 10(b), with two instances of the `ShiftLossyFIFO1` circuit of Figure 11(b) spliced in. In this version of our alternating producers and consumer, these three processes are partially decoupled: each producer runs at its own pace, never having to wait for any of the other two processes. Every take by the consumer, always obtains and empties the latest value produced by its respective producer. If the consumer runs slower than a producer, the excess data that they produce is lost in the producer's respective `ShiftLossyFIFO1`, which allows the consumer to effectively "sample" the data generated by this producer. If the consumer runs faster than a producer, it will block on its respective empty `ShiftLossyFIFO1` until a new value becomes available.

4.6 Dataflow Variable

The Reo circuit in Figure 13 implements the behavior of a dataflow variable. It uses two instances of the `ShiftLossyFIFO1` connector shown Figure 11(b), to build a connector with a single input and a single output nodes. Initially, the buffers of its `ShiftLossyFIFO1` 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 `ShiftLossyFIFO1` 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.

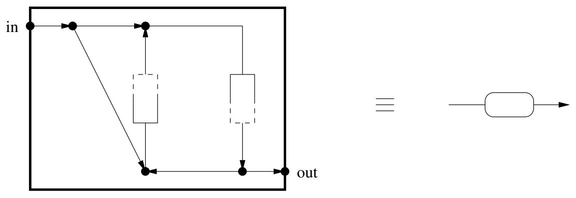

Fig. 13. Dataflow variable

The variable circuit in Figure 13 is also very frequently useful as a connector in construction of more complex circuits. Therefore, it makes sense to have a shorthand graphical symbol to designate it with as well. The symbol shown on the right-hand side of Figure 13 is the what we use to represent this circuit, and also take the liberty to refer to it as a `Variable` "channel", or just a "variable" for short. This symbol is intentionally similar to that of a regular `FIFO1` channel, because the behavior of this circuit closely resembles that of a regular `FIFO1` channel. 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.

4.7 Fully Decoupled Alternating Producers and Consumer

Figure 14(a) shows how the variable circuit of Figure 13 can be used to construct a version of the example in Figure 5, where the producer and the consumer are fully decoupled from one another. Initially, the variable contains no value, and therefore, the consumer has no choice but to wait for the producer to place its first value into the variable. After that, neither the producer, nor the consumer ever has to wait for the other one. Each can work at its own pace and write to or take from the connector. Every write by the producer replaces the current contents of the variable, and every take by the consumer obtains a copy of the current value of the variable, which always contains the most recent value produced.

The connector in Figure 14(b) is a small variation of the Reo circuit in Figure 10(b), with two instances of the variable circuit of Figure 13 spliced in. In this version of our alternating producers and consumer, these three processes are fully decoupled: each can produce and consume at its own pace, never having to wait for any of the other two. Every take by the consumer, always obtains the latest value produced by its respective producer. If the consumer runs slower than a producer, the excess data is lost in the producer's respective variable,

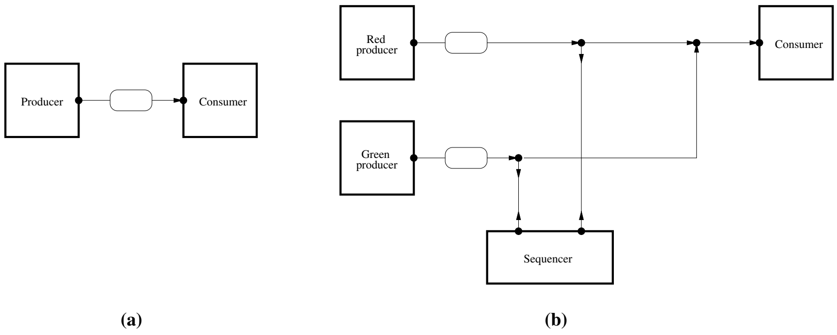

Fig. 14. Fully decoupled producers and consumer

and the consumer will effectively “sample” the data generated by this producer. If the consumer runs faster than a producer, it will read (some of) the values of this producer multiple times.

4.8 Flexibility

Figures 9(a), 10(b), 12(b), and 14(b) show four different connectors, each imposing a different protocols for the coordination of two alternating producers and a consumer. The exact same producers and consumer processes can be combined with any of these circuits to yield different applications. It is instructive to compare the ease with which this is accomplished in our interaction-centric world, with the effort involved in modifying the action-centric incarnations of this same example in Figures 3 and 4, which correspond to the protocol of the circuit in Figure 10(b), in order to achieve the behavior induced by the circuit in Figure 9(a), 12(b), or 14(b).

For the sake of completeness, the behavior of the protocol in Figures 1 corresponds to the behavior of the connector in Figure 15. Just as in the case of the program in Figures 1, this connector allows the producers at nodes A and B alternate and over-write each other in buffer of the `ShiftLossyFIFO1`. The consumer at C can obtain only the latest value produced by either of the producers.

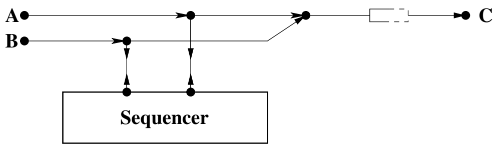

Fig. 15. Alternating and over-writing

The Reo connector binding a number of distributed processes, such as Web services, can even be "hot-swapped" while the application runs, without the knowledge or the involvement of the engaged processes. A prototype platform to demonstrate this capability is available at [3].

5 Semantics

Reo allows arbitrary user-defined channels as primitives; arbitrary mix of synchrony and asynchrony; and relational constraints between input and output. This makes Reo more expressive than, e.g., dataflow models, Kahn networks, synchronous languages, stream processing languages, workflow models, and Petri nets. On the other hand, it makes the semantics of Reo quite non-trivial.

Various models for the formal semantics of Reo have been developed, each to serve some specific purposes. In the rest of this section, we briefly describe the main ones.

5.1 Timed Data Streams

The first formal semantics of Reo was formulated based on the coalgebraic model of stream calculus [95,94,96]. In this semantics, the behavior of every connector (channel or more complex circuit) and every component is given as a (maximal) relation on a set of *timed-data-streams* [24]. This yields an expressive compositional semantics for Reo where coinduction is the main definition and proof principle to reason about properties involving both data and time streams. The timed-data-stream model serves as the reference semantics for Reo.

Table 1. TDS Semantics of Reo primitives

`Sync`	$\langle\alpha,a\rangle Sync\langle\beta,b\rangle \equiv \alpha=\beta \wedge a=b$
`LossySync`	$\langle\alpha,a\rangle\ \texttt{LossySync}\ \langle\beta,b\rangle \equiv \begin{cases}\beta(0)=\alpha(0)\wedge\langle\alpha',a'\rangle\ \texttt{LossySync}\ \langle\beta',b'\rangle & \text{if } a(0)=b(0)\\ \langle\alpha',a'\rangle\ \texttt{LossySync}\ \langle\beta,b\rangle & \text{if } a(0)<b(0)\end{cases}$
empty `FIFO1`	$\langle\alpha,a\rangle FIFO1\langle\beta,b\rangle \equiv \alpha=\beta \wedge a<b<a'$
`FIFO1` initialized with x	$\langle\alpha,a\rangle FIFO1(x)\langle\beta,b\rangle \equiv \alpha=x.\beta \wedge b<a<b'$
`SyncDrain`	$\langle\alpha,a\rangle SyncDrain\langle\beta,b\rangle \equiv a=b$
`AsyncDrain`	$\langle\alpha,a\rangle SyncDrain\langle\beta,b\rangle \equiv a\neq b$
`Filter(P)`	$\langle\alpha,a\rangle\ \texttt{Filter}(P)\ \langle\beta,b\rangle \equiv \begin{cases}\beta(0)=\alpha(0)\wedge b(0)=a(0)\wedge\langle\alpha',a'\rangle\ \texttt{Filter}(P)\ \langle\beta',b'\rangle & \text{if } \alpha(0)\ni P\\ \langle\alpha',a'\rangle\ \texttt{Filter}(P)\ \langle\beta,b\rangle & \text{otherwise}\end{cases}$
Merge	$Mrg(\langle\alpha,a\rangle,\langle\beta,b\rangle;\langle\gamma,c\rangle) \equiv \begin{cases}\alpha(0)=\gamma(0)\wedge a(0)=c(0)\wedge Mrg(\langle\alpha',a'\rangle,\langle\beta,b\rangle;\langle\gamma',c'\rangle) & \text{if } a(0)<b(0)\\ \beta(0)=\gamma(0)\wedge b(0)=c(0)\wedge Mrg(\langle\alpha,a\rangle,\langle\beta',b'\rangle;\langle\gamma',c'\rangle) & \text{if } a(0)>b(0)\end{cases}$
Replicate	$Rpl(\langle\alpha,a\rangle;\langle\beta,b\rangle,\langle\gamma,c\rangle) \equiv \alpha=\beta\wedge\alpha=\gamma\wedge a=b\wedge a=c$

A *stream* over a set X is an infinite sequence of elements $x \in X$. The set of *data streams* DS consists of all streams over an uninterpreted set of *Data* items. A *time stream* is a monotonically increasing sequence of non-negative real numbers. The set TS represents all time streams[2]. A *Timed Data Stream* (TDS) 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 observation of the data item $\alpha(i)$ occurs at the time moment $a(i)$. We use a' to represent the tail of a stream a, i.e., the stream obtained after removing the first element of a; and $x.a$ to represent the stream whose first element is x and whose tail is a.

Table 1 shows the TDS semantics of the primitive channels in Figure 6, as well as that of the merge and replication behavior inherent in Reo nodes. The semantics for every primitive is expressed as a binary (in the case of channels) or ternary (for the merger and the replicator) relation on timed-data-streams that represent the observations at their respective source and sink ends. We can use relational composition to combine the semantics of these primitives to obtain the semantics of more complex connectors. For instance, by composing the relation that defines a binary merger in Table 1 with that of another, we can obtain the semantics for a ternary merger. Thus, the semantics of an m-ary sink node in Reo can be obtained as the composition of $m - 1$ binary mergers. Analogously, the semantics of and n-are source node in Reo can be obtained as the composition of $n - 1$ binary replicators. The semantics of a Reo mixed node with m coincident sink and n coincident source channel ends is obtained as the relational composition of $m - 1$ binary mergers and $n - 1$ binary replicators.

The semantics of a Reo circuit is the relational composition of the relations that represent the semantics of its constituents (including the merge and replication inherent in its nodes). This compositional construction for instance, yields

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

as the semantics of the circuit in Figure 11(a).

5.2 Constraint Automata

Constraint automata provide an operational model for the semantics of Reo circuits [31]. The states of an automaton represent the configurations of its corresponding circuit (e.g., the contents of the FIFO channels), while the transitions encode its maximally-parallel stepwise behavior. The transitions are labeled with the maximal sets of nodes on which dataflow occurs simultaneously, and a data constraint (i.e., boolean condition for the observed data values). For example, Figure 16 shows the constraint automata semantics for some of the common Reo primitives.

[2] The real numbers that appear in a time stream must also satisfy an additional technical condition to prevent Zeno's paradox, but for simplicity, we ignore this condition here.

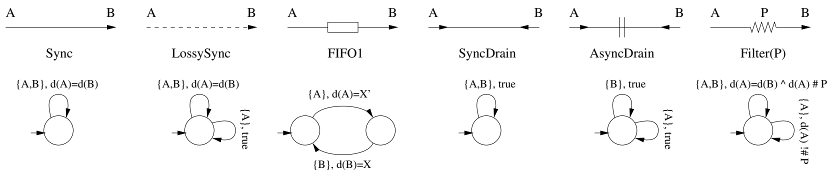

Fig. 16. Constraint automata of some typical Reo Channels

The constraint automaton for the Sync channel consists of a single state. It has only a single transition, labeled by the pair of *synchronization constraint*, and *data constraint*. The synchronization constraint $\{A, B\}$ states that this transition is possible iff both nodes A and B can *fire* synchronously (i.e., atomically), allowing their respective pending I/O operations to succeed. The data constraint $d(A) = d(B)$ states that this transition is possible iff the data observed at node A is identical to the data observed at node B. Because these two nodes are respectively the source and the sink nodes (of the Sync channel), this data constraint requires a transfer of data from A to B.

The constraint automaton for the LossySync channel in fact expresses the semantics of a *nondeterministic* LossySync channel, *not* that of our *context sensitive* LossySync described in Section 3. The difference is significant, but it is not important for our purposes in this paper.[3] This automaton has a single state and two transitions. One of these transitions is identical to that of the Sync channel, modeling its identical behavior. The other, labeled by $\{A\}$, *true* simply states that the automaton can make this transition iff A can fire by itself and imposes no constraint of the data of A: this data is lost.

The constraint automaton for the FIFO1 channel has two states, representing its empty (initial) and full states. To simplify our presentation, we consider a variant of constraint automata that allow states to have local memory variables. The label $\{A\}, d(A) = X'$ of the transition that takes the automaton from its empty to its full state allows it to make this transition iff node A can fire by itself, and the *new value* of the memory variable X in the target state (identified by X' in the data constraint) is the same as the data value observed on node A: the value obtained from the source node A gets assigned to the X variable of the target state to satisfy this constraint. The label $\{B\}, d(B) = X$ of the transition that takes the automaton from its full to its empty state allows it to make this transition iff node B can fire by itself, and the value of the memory variable X in the source state (identified by X in the data constraint) is the same as the data value observed on node B: the value of the X variable of the source state is dispensed through the sink node B to satisfy this data constraint.

[3] In fact, constraint automata do not have the expressiveness required to directly represent context sensitivity. Other more expressive semantic models, including more sophisticated automata models, have been devised for this purpose [35,44]. A recent work shows that, although constraint automata cannot directly represent context sensitivity, it is possible to *encode* context sensitivity using constraint automata as well [56,70].

The constraint automaton for the SyncDrain channel has a single state and a single transition, whose constraints require its ends to fire synchronously ($\{A, B\}$), but imposes no constraints (*true*) on their data. Because these are both source ends, their data are simply lost.

The constraint automaton for the AsyncDrain channel has a single state and two transitions, each of which allow it to fire and lose the data obtained through one of its ends (but never both synchronously).

The constraint automaton for the Filter(P) channel has a single state and two transitions. If source node A can fire and its data value does not match the filter pattern P, then the data value of A is simply lost. If the data value available on the source node A matches the filter pattern P, then the only possible transition is one similar to that of the Sync channel, by which the data value of A is transferred to the sink node B.

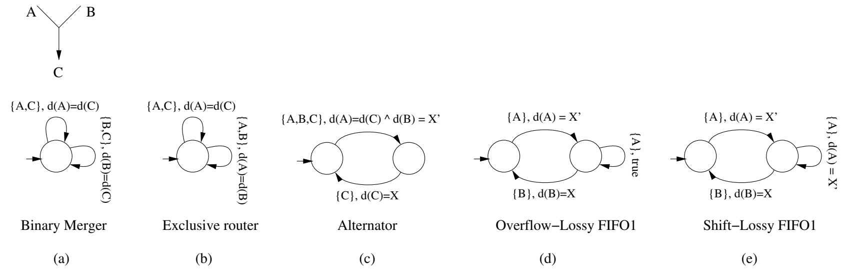

Fig. 17. Constraint automata of a binary merger and some example connectors

The semantics of a Reo circuit is derived by composing the constraint automata of its constituents, through a special form of synchronized product of automata, which automatically accommodates the replication semantics of Reo nodes [31]. The nondeterministic n-ary merge semantics inherent in Reo nodes needs to be made explicit as a (product) composition of $n-1$ nondeterministic binary merge primitives. Figure 17(a) shows the constraint automaton for a nondeterministic binary merge primitive.

Figure 17(b) shows the constraint automaton representing the semantics of the exclusive router Reo circuit of Figure 11(a), which is obtained as the product of the constraint automata of its constituents: 5 Sync channels, 2 LossySync channels, a SyncDrain channel, and a merger.

Figure 17(c) shows the constraint automaton representing the semantics of the alternator circuit of Figure 9(a), obtained as the product of the constraint automata of its constituent Sync channel, SyncDrain channel, FIFO1 channel, and merger.

Figure 17(d) shows the constraint automaton representing the semantics of an *overflow lossy* connector, which can be easily composed by connecting the sink end of a LossySync to the source end of a FIFO1. Although this *is* the semantics that must be obtained, the product of simple constraint automata in Figure 16 does *not* yield this automaton. This automaton can be obtained using more sophisticated variants of constraint automata [35,44], or an encoding technique [56] which can handle context sensitivity.

Figure 17(e) shows the constraint automaton representing the semantics of the `ShiftLossyFIFO1` circuit of Figure 11(b), which is obtained as the product of the constraint automata of its constituents.

Constraint automata have been used for the verification of protocols through model-checking [7,62,34,28,61,30,29,48]. Results on equivalence and containment of the languages of constraint automata [31] and failure based equivalences [54] provide opportunities for analysis and optimization of Reo circuits.

A constraint automaton essentially captures all behavior alternatives of a Reo connector. Therefore, it can be used to generate a state-machine implementing the behavior of Reo connectors, in a chosen target language, such as Java or C. The constraint automata semantics of Reo is used to generate executable code for Reo [18].

Variants of the constraint automata model have been devised to capture time-sensitive behavior [14,58,59], probabilistic behavior [26], stochastic behavior [32], context sensitive behavior [35,44,52], fairness [53,36], resource sensitivity [79], and the QoS aspects [80,16,17,87,86] of Reo connectors and composite systems.

5.3 Connector Coloring

The Connector Coloring (CC) model describes the behavior of a Reo circuit in terms of the detailed dataflow behavior of its constituent channels and nodes [41]. The semantics of a Reo circuit is the set of all of its dataflow alternatives. Each such alternative is a consistent composition of the dataflow alternatives of each of its constituent channels and nodes, expressed in terms of (solid and dashed) *colors* that represent the basic flow and no-flow alternatives.

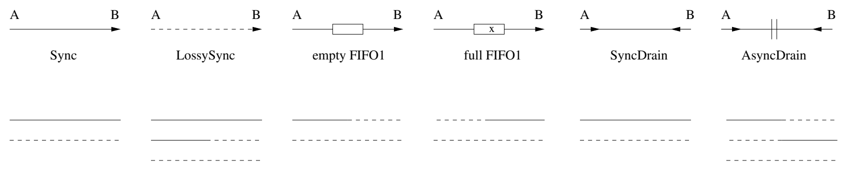

Fig. 18. Connector Coloring semantics of some typical Reo Channels

Figure 18 shows the two-color semantics of some common Reo primitives. The `Sync` channel has two alternative colorings, each representing one possible behavior: either flow on both of its ends (the solid line) or no flow on both ends (the dashed line). The (nondeterministic) `LossySync` has three alternative colorings: it either behaves as the `Sync` channel (the full solid and the full dashed lines), or it allows flow at its source end, with no flow at its sink end (the half-solid-half-dashed line). A `FIFO1` channel has two sets of colorings, one for each of its two states: empty and full. In its empty state, it can allow flow only at its source end (with no flow at its sink), after which it becomes full. In its full state, it can allow flow only at its sink node (with no flow at its source), which makes it empty. A `SyncDrain` channel has the same coloring as a `Sync` channel: it can allow flow only through both of its ends simultaneously. An `AsyncDrain` allows flow through only one of its ends at a time.

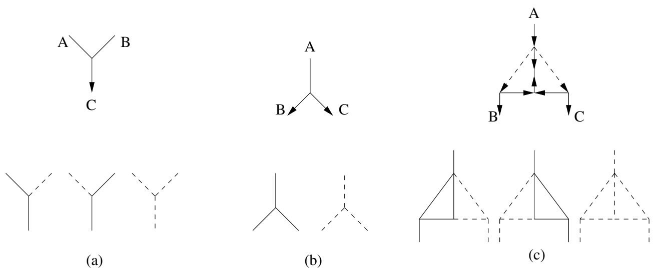

Fig. 19. Connector Coloring semantics for merger, replicator, and exclusive router

To express the semantics of a Reo circuit, the replicator and the merger behavior inherent in Reo nodes must also be explicitly modeled as colorings. Figure 19(a) shows the three alternatives for the behavior of a merger: the merger nondeterministically chooses to allow flow either theough its left source and sink, or through its right source and sink, or there is no flow on any of its ends. Figure 19(b) shows the two alternatives for the behavior of a replicator: either there is flow on its source and both sinks, or there is no flow through any of its nodes at all.

The coloring semantics of a Reo circuit can be composed out of the coloring alternatives of its constituents, subject to the obvious requirement that each node in the circuit can either have flow or not, and therefore, the colors of the behavior alternatives of all constituents that coincide on a node must be the same: either dashed or solid. For example, the coloring alternatives of the exclusive router circuit of Figure 11(a) is obtained by matching the alternative colors of its constituent channels, replicators, and merger, as shown in Figure 19(c). As expected, this circuit as a whole allows flow through either its right-hand side, or its left-hand side, exclusively, or there is no flow through the circuit at all.

A more sophisticated model using three colors is necessary to capture the context sensitive behavior of primitives such as the `LossySync` channel. The CC model is primarily used in the implementation of a visualization tool that produces Flash animations depicting the behavior of a connector [44,91,75]. Connector coloring and constraint automata are related [55]. It has been shown that it is possible to encode context sensitive behavior in the two-color CC model as well, using hypothetical extra nodes [56].

Finding a consistent coloring for a circuit amounts to constraint satisfaction. Constraint solving techniques [10,102] have been applied using the CC model to search for a valid global behavior of a given Reo connector [42,43]. In this approach, each connector is considered as a set of constraints, representing the colors of its individual constituents, where valid solutions correspond to a valid behavior for the current step. Distributed constraint solving techniques can be used to adapt this constraint based approach for distributed environments.

The CC model is at the center of the distributed implementation of Reo [2,91,93] where several engines, each executing a part of the same connector, run on different remote hosts. A distributed protocol based on the CC model guarantees that all engines running the various parts of the connector agree to collectively manifest one of its legitimate behavior alternatives.

5.4 Other Models

Other formalisms have also been used to investigate the various aspects of the semantics of Reo. Plotkin's style of Structural Operational Semantics (SOS) is followed in [89] for the formal semantics of Reo. This semantics was used in a proof-of-concept tool developed in the rewriting logic language of Maude, using the simulation toolkit.

The Tile Model [47] semantics of Reo offers a uniform setting for representing not only the ordinary dataflow execution of Reo connectors, but also their dynamic reconfigurations [15]. An abstraction of the constraint automata is used in [74] to serve as a common semantics for Reo and Petri nets. The application of intuitionistic temporal linear logic (ITLL) as a basis for the semantics of Reo is studied in [38], which also shows the close semantic link between Reo and the zero-safe variant of Petri nets. A comparison of Orc [85,60] and Reo appears in [92], and the authors of [99] compare Reo with ARC and PBRD coordination models.

The semantics of Reo has also been formalized in the Unifying Theories of Programming (UTP) [51]. The UTP approach provides a family of algebraic operators that interpret the composition of Reo connectors more explicitly than in other approaches [81]. This semantic model can be used for proving properties of connectors, such as equivalence and refinement relations between connectors and as a reference document for developing tool support for Reo. The UTP semantics for Reo opens the possibility to integrate reasoning about Reo with reasoning about component specifications/implementations in other languages for which UTP semantics is available. The UTP semantics of Reo has been used for fault-based test case generation [9].

Automatic translation of an automata-based semantics of Reo into its equivalent process algebraic specification is the basis of another input-output conformance testing of protocols specified in Reo [70].

Reo offers operations to dynamically reconfigure the topology of its coordinator circuits, thereby changing the coordination protocol of a running application. A semantic model for Reo cognizant of its reconfiguration capability, a logic for reasoning about reconfigurations, together with its model checking algorithm, are presented in [39]. Graph transformation techniques have been used in combination with the connector coloring model to formalize the dynamic reconfiguration semantics of Reo circuits triggered by dataflow [64,63,76,75].

6 Tools

Tool support for Reo consists of a set of Eclipse plug-ins that together comprise the Extensible Coordination Tools (ECT) visual programming environment [3]. The Reo graphical editor supports drag-and-drop graphical composition and editing of Reo circuits. This editor also serves as a bridge to other tools, including animation and code generation plug-ins. The animation plug-in automatically generates a graphical animation of the flow of data in a Reo circuit, which provides an intuitive insight into their behavior through visualization of how they work. This tool maps the colors of the CC semantics to visual representations in the animations, and represents the movement of data through the connector [44,91].

Another graphical editor in ECT supports drag-and-drop construction and editing of constraint automata and its variants. It includes tools to perform product and hiding on constraint automata for their composition. A converter plug-in automatically generates the CA model of a Reo circuit.

Several model checking tools are available for analyzing Reo. The Vereofy model checker, integrated in ECT, is based on constraint automata [7,34,62,27,28,61,30,29,48]. Vereofy supports two input languages: (1) the Reo Scripting Language (RSL) is a textual language for defining Reo circuits, and (2) the Constraint Automata Reactive Module Language (CARML) is a guarded command language for textual specification of constraint automata. Properties of Reo circuits can be specified for verification by Vereofy in a language based on Linear Temporal Logic (LTL), or on a variant of Computation Tree Logic (CTL), called Alternating-time Stream Logic (ASL). Vereofy extends these logics with regular expression constructs to express data constraints. Translation of Reo circuits and constraint automata into RSL and CARML is automatic, and the counter-examples found by Vereofy can automatically be mapped back into the ECT and displayed as Reo circuit animations.

Timed Constraint Automata (TCA) were devised as the operational semantics of timed Reo circuits [14]. A SAT-based bounded model checker exists for verification of a variant of TCA [58,59], although it is not yet fully integrated in ECT. It represents the behavior of a TCA by formulas in propositional logic with linear arithmetic, and uses a SAT solver for their analysis. A tool is available to translate (timed) Reo circuits into models for verification using the Uppaal model checker.

Another means for verification of Reo is made possible by a transformation bridge into the mCRL2 toolset [4,49]. The mCRL2 verifier relies on the parameterized boolean equation system (PBES) solver to encode model checking problems, such as verifying first-order modal-calculus formulas on linear process specifications. An automated tool integrated in ECT translates Reo models into mCRL2 and provides a bridge to its tool set. This translation and its application for the analysis of workflows modeled in Reo are discussed in [67,72,71]. Through mCRL2, it is possible to verify the behavior of timed Reo circuits, or Reo circuits with more elaborate data-dependent behavior than Vereofy supports. The resulting labeled transformation systems can also be used for analysis by a number of tools in the CADP tool set [1].

A CA code generator plug-in produces executable Java code from a constraint automaton as a single sequential thread. A C/C++ code generator is under development. In this setting, components communicate via put and get operations on so-called *SyncPoints* that implement the semantics of a constraint automaton port, using common concurrency primitives. The tool also supports loading constraint automata descriptions at runtime, useful for deploying Reo coordinators in Java application servers, e.g., Tomcat, for applications such as mashup execution [66,78].

A distributed implementation of Reo exists [2] as a middleware in the actor-based language Scala [90], which generates Java source code. A preliminary integration of this distributed platform into ECT provides the basic functionality for distributed deployment through extensions of the Reo graphical editor [91].

A set of ECT plug-in tools are under development to support coordination and composition of Web Services using Reo. ECT plug-ins are available for automatic conversion of coordination and concurrency models expressed as UML sequence diagrams [21,22], UML activity diagrams, BPMN diagrams [19], and BPEL source code into Reo circuits [37].

Tools are integrated in ECT for automatic generation of Quantified Intentional Constraint Automata (QIA) from Reo circuits annotated with QoS properties, and subsequent automatic translation of the resulting QIA to Markov Chain models [16,17,87,86]. A bridge to Prism [5] allows further analysis of the resulting Markov chains [23]. Of course, using Markov chains for the analysis of the QoS properties of a Reo circuit (and its environment) is possible only when the stochastic variables representing those QoS properties can be modeled by exponential distributions. The QIA, however, remain oblivious to the (distribution) types of stochastic variables. A discrete event simulation engine integrated in ECT supports a wide variety of more general distributions for the analysis of the QoS properties of Reo circuits [57,100].

Based on algebraic graph transformations, a reconfiguration engine is available as an ECT plug-in that supports dynamic reconfiguration of distributed Reo circuits triggered by dataflow [18,63,75]. It currently works with the Reo animation engine in ECT, and will be integrated in the distributed implementation of Reo.

7 Concluding Remarks

Action and interaction offer dual perspectives on concurrency. Execution of actions involving shared resources by independent processes that run concurrently, induces pairings of those actions, along with an ordering of those pairs, that we commonly refer to as interaction. Dually, interaction can be seen as an external relation that constrains the pairings of the actions of its engaged processes and their ordering. The traditional action-centric models of concurrency generally make interaction protocols intangible by-products, implied by nebulous specifications scattered throughout the bodies of their engaged processes. Specification, manipulation, and analysis of such protocols are possible only indirectly, through specification, manipulation, and analysis of those scattered actions, which is often made even more difficult by the entanglement of the data-dependent control flow that surrounds those actions. The most challenging aspect of a concurrent system is *what* its interaction protocol does. In contrast to the *how* which an imperative programming language specifies, declarative programming, e.g., in functional and constraint languages, makes it easier to directly specify, manipulate, and analyze the properties of *what* a program does, because *what* is precisely what they express. Analogously, in an interaction-centric model of concurrency, interaction protocols become tangible first-class constructs that exist explicitly as (declarative) constraints outside and independent of the processes that they engage. Specification of interaction protocols as declarative constraints makes them easier to manipulate and analyze directly, and makes it possible to compose interaction protocols and reuse them.

The coordination language Reo is a premier example of a formalism that embodies an interaction-centric model of concurrency. We used examples of Reo circuits to illustrate the flavor programming pure interaction protocols. Expressed as explicit declarative constraints, protocols espouse exogenous coordination. Our examples showed the utility of exogenous coordination in yielding loosely-coupled flexible systems whose components and protocols can be easily modified, even at run time. We described a set of prototype support tools developed as plug-ins to provide a visual programming environment within the framework of Eclipse, and presented an overview of the formal foundations of the work behind these tools.

A dragon lives forever, but not so little boys. Nevertheless, the ecology of today's society has left no secluded cave for our Puff to sadly slip into. The protocols that our magic dragon manifests in its wake as it frolics through the lines of code of concurrent applications will likely touch many aspects of the daily life of every adult Jackie Paper. We have grown to know our magic dragon well through the intimacy of the childhood games we played with it. Scaled up versions of those games have become integral to the proper functioning of our lives as grownups. Wish as we may to make way for other toys, we cannot abandon this magic dragon any more. We need to develop concise languages to directly communicate with our dragon in concrete terms of a structured dialog that explicitly conveys the constraints of acceptable behavior in the context of our requirements. Reo is a particular dialect of one such language.

References

1. 7CADP home page, `http://www.inrialpes.fr/vasy/cadp/`
2. Distributed Reo, `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Redrum/BigPicture`
3. Extensible Coordination Tools home page, `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools`
4. mCRL2 home page, `http://www.mcrl2.org`
5. Prism, `http://www.prismmodelchecker.org`
6. Reo home page, `http://reo.project.cwi.nl`
7. Vereofy home page, `http://www.vereofy.de/`

8. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
9. Aichernig, B.K., Arbab, F., Astefanoaei, L., de Boer, F.S., Meng, S., Rutten, J.J.M.M.: Fault-based test case generation for component connectors. In: Chin, W.-N., Qin, S. (eds.) TASE, pp. 147–154. IEEE Computer Society (2009)
10. Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)
11. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14(3), 329–366 (2004)
12. Arbab, F.: Abstract Behavior Types: a foundation model for components and their composition. Sci. Comput. Program. 55(1-3), 3–52 (2005)
13. Arbab, F., Aştefănoaei, L., de Boer, F.S., Dastani, M.M., Meyer, J.-J., Tinnermeier, N.: Reo Connectors as Coordination Artifacts in 2APL Systems. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 42–53. Springer, Heidelberg (2008)
14. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. Software and System Modeling 6(1), 59–82 (2007)
15. Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U.: Tiles for Reo. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 37–55. Springer, Heidelberg (2009)
16. Arbab, F., Chothia, T., Meng, S., Moon, Y.-J.: Component connectors with QoS guarantees. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 286–304. Springer, Heidelberg (2007)
17. Arbab, F., Chothia, T., van der Mei, R., Meng, S., Moon, Y.-J., Verhoef, C.: From coordination to stochastic models of QoS. In: Field, Vasconcelos (eds.) [45], pp. 268–287
18. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y.-J., Proença, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. Tool demo session at FACS 2008 (2008)
19. Arbab, F., Kokash, N., Meng, S.: Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA. CCIS, vol. 17, pp. 108–123. Springer, Heidelberg (2008)
20. Arbab, F., Mavaddat, F.: Coordination Through Channel Composition. In: Arbab, F., Talcott, C. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 22–39. Springer, Heidelberg (2002)
21. Arbab, F., Meng, S.: Synthesis of Connectors From Scenario-Based Interaction Specifications. In: Chaudron, M.R.V., Szyperski, C.A., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 114–129. Springer, Heidelberg (2008)
22. Arbab, F., Meng, S., Baier, C.: Synthesis of Reo circuits from scenario-based specifications. Electr. Notes Theor. Comput. Sci. 229(2), 21–41 (2009)
23. Arbab, F., Meng, S., Moon, Y.-J., Kwiatkowska, M.Z., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: van Vliet, H., Issarny, V. (eds.) ESEC/SIGSOFT FSE, pp. 287–288. ACM (2009)
24. Arbab, F., Rutten, J.J.M.M.: A Coinductive Calculus of Component Connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
25. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press (1990)
26. Baier, C.: Probabilistic models for Reo connector circuits. Journal of Universal Computer Science 11(10), 1718–1748 (2005)

27. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal Verification for Components and Connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009)
28. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field, Vasconcelos (eds.) [45], pp. 247–267
29. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S., Leister, W.: Design and Verification of Systems with Exogenous Coordination Using Vereofy. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010)
30. Baier, C., Klein, J., Klüppelholz, S.: Modeling and Verification of Components and Connectors. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 114–147. Springer, Heidelberg (2011)
31. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)
32. Baier, C., Wolf, V.: Stochastic Reasoning About Channel-Based Component Connectors. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 1–15. Springer, Heidelberg (2006)
33. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60, 109–137 (1984)
34. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. Electr. Notes Theor. Comput. Sci. 215, 209–226 (2008)
35. Bonsangue, M.M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Field, Vasconcelos (eds.) [45], pp. 184–203
36. Bonsangue, M.M., Izadi, M.: Automata based model checking for Reo connectors. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 260–275. Springer, Heidelberg (2010)
37. Changizi, B., Kokash, N., Arbab, F.: A unified toolset for business process model formalization. In: Proc. of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2010 (2010); satellite event of ETAPS
38. Clarke, D.: Coordination: Reo, Nets, and Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 226–256. Springer, Heidelberg (2008)
39. Clarke, D.: A basic logic for reasoning about connector reconfiguration. Fundam. Inform. 82(4), 361–390 (2008)
40. Clarke, D., Costa, D., Arbab, F.: Modelling Coordination in Biological Systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 9–25. Springer, Heidelberg (2006)
41. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Sci. Comput. Program. 66(3), 205–225 (2007)
42. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Deconstructing Reo. Electr. Notes Theor. Comput. Sci. 229(2), 43–58 (2009)
43. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Sci. Comput. Program. 76(8), 681–710 (2011)
44. Costa, D.: Formal Models for Context Dependent Connectors for Distributed Software Components and Services. PhD thesis, Vrije Universiteit Amsterdam (2010), `http://dare.ubvu.vu.nl//handle/1871/16380`
45. Field, J., Vasconcelos, V.T. (eds.): COORDINATION 2009. LNCS, vol. 5521, pp. 225–246. Springer, Heidelberg (2009)

46. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer, Heidelberg (1999)
47. Gadducci, F., Montanari, U.: The tile model. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, pp. 133–166. MIT Press, Boston (2000)
48. Grabe, I., Jaghoori, M.M., Aichernig, B.K., Baier, C., Blechmann, T., de Boer, F.S., Griesmayer, A., Johnsen, E.B., Klein, J., Klüppelholz, S., Kyas, M., Leister, W., Schlatte, R., Stam, A., Steffen, M., Tschirner, S., Xuedong, L., Yi, W.: Credo methodology: Modeling and analyzing a peer-to-peer system in credo. Electr. Notes Theor. Comput. Sci. 266, 33–48 (2010)
49. Groote, J.F., Mathijssen, A., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.: The formal specification language mCRL2. In: Brinksma, E., Harel, D., Mader, A., Stevens, P., Wieringa, R. (eds.) MMOSS. Dagstuhl Seminar Proceedings, vol. 06351. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl (2006)
50. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
51. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall, London (1998)
52. Izadi, M., Bonsangue, M.M., Clarke, D.: Modeling component connectors: Synchronisation and context-dependency. In: Cerone, A., Gruner, S. (eds.) SEFM, pp. 303–312. IEEE Computer Society (2008)
53. Izadi, M., Bonsangue, M.M., Clarke, D.: Büchi automata for modeling component connectors. Software and System Modeling 10(2), 183–200 (2011)
54. Izadi, M., Movaghar, A.: Failure-based equivalence of constraint automata. Int. J. Comput. Math. 87(11), 2426–2443 (2010)
55. Jongmans, S.-S.T.Q., Arbab, F.: Correlating formal semantic models of Reo connectors: Connector coloring and constraint automata. In: Silva, A., Bliudze, S., Bruni, R., Carbone, M. (eds.) ICE. EPTCS, vol. 59, pp. 84–103 (2011)
56. Jongmans, S.-S.T.Q., Krause, C., Arbab, F.: Encoding context-sensitivity in Reo into non-context-sensitive semantic models. In: Meuter, Roman (eds.) [82], pp. 31–48
57. Kanters, O.: QoS analysis by simulation in Reo (2010)
58. Kemper, S.: SAT-based Verification for Timed Component Connectors. Electr. Notes Theor. Comput. Sci. 255, 103–118 (2009)
59. Kemper, S.: Compositional Construction of Real-Time Dataflow Networks. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 92–106. Springer, Heidelberg (2010)
60. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The Orc Programming Language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
61. Klein, J., Klüppelholz, S., Stam, A., Baier, C.: Hierarchical Modeling and Formal Verification. An Industrial Case Study Using Reo and Vereofy. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 228–243. Springer, Heidelberg (2011)
62. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Electr. Notes Theor. Comput. Sci. 175(2), 19–37 (2007)
63. Koehler, C., Arbab, F., de Vink, E.P.: Reconfiguring Distributed Reo Connectors. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 221–235. Springer, Heidelberg (2009)

64. Koehler, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of Reo connectors triggered by dataflow. In: Ermel, C., Heckel, R., de Lara, J. (eds.) Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008), vol. 10, pp. 1–13 (2008); Home Page, `http://www.easst.org/eceasst/`, ECEASST ISSN 1863-2122
65. Koehler, C., Lazovik, A., Arbab, F.: ReoService: Coordination modeling tool. In: Krämer, et al. (eds.) [73], pp. 625–626
66. Koehler, C., Lazovik, A., Arbab, F.: ReoService: Coordination Modeling Tool. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 625–626. Springer, Heidelberg (2007)
67. Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: SAC 2010: Proc. of the 2010 ACM Symposium on Applied Computing, pp. 2406–2413. ACM, New York (2010)
68. Kokash, N., Arbab, F.: Formal behavioral modeling and compliance analysis for service-oriented systems. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 21–41. Springer, Heidelberg (2009)
69. Kokash, N., Arbab, F.: Applying Reo to service coordination in long-running business transactions. In: Shin, S.Y., Ossowski, S. (eds.) SAC, pp. 1381–1382. ACM (2009)
70. Kokash, N., Arbab, F., Changizi, B., Makhnist, L.: Input-output conformance testing for channel-based service connectors. In: Aceto, L., Mousavi, M.R. (eds.) PACO. EPTCS, vol. 60, pp. 19–35 (2011)
71. Kokash, N., Krause, C., de Vink, E.P.: Verification of Context-Dependent Channel-Based Service Models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 21–40. Springer, Heidelberg (2010)
72. Kokash, N., Krause, C., de Vink, E.P.: Time and data-aware analysis of graphical service models in Reo. In: Fiadeiro, J.L., Gnesi, S., Maggiolo-Schettini, A. (eds.) SEFM, pp. 125–134. IEEE Computer Society (2010)
73. Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.): ICSOC 2007. LNCS, vol. 4749. Springer, Heidelberg (2007)
74. Krause, C.: Integrated structure and semantics for Reo connectors and Petri nets. In: ICE 2009: Proc. 2nd Interaction and Concurrency Experience Workshop. Electronic Proceedings in Theoretical Computer Science, vol. 12, p. 57 (2009)
75. Krause, C.: Reconfigurable Component Connectors. PhD thesis, Leiden University (2011), `https://openaccess.leidenuniv.nl/handle/1887/17718`
76. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Sci. Comput. Program. 76(1), 23–36 (2011)
77. Lazovik, A., Arbab, F.: Using Reo for service coordination. In: Krämer, et al. (eds.) [73], pp. 398–403
78. Maraikar, Z., Lazovik, A.: Reforming mashups. In: Proceedings of the 3rd European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2008). Imperial College London (June 2008)
79. Meng, S., Arbab, F.: On Resource-Sensitive Timed Component Connectors. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 301–316. Springer, Heidelberg (2007)
80. Meng, S., Arbab, F.: QoS-driven service selection and composition. In: Billington, J., Duan, Z., Koutny, M. (eds.) ACSD, pp. 160–169. IEEE (2008)
81. Meng, S., Arbab, F.: Connectors as designs. Electr. Notes Theor. Comput. Sci. 255, 119–135 (2009)

82. De Meuter, W., Roman, G.-C. (eds.): COORDINATION 2011. LNCS, vol. 6721. Springer, Heidelberg (2011)
83. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
84. Milner, R.: Elements of interaction - turing award lecture. Commun. ACM 36(1), 78–89 (1993)
85. Misra, J., Cook, W.R.: Computation orchestration. Software and System Modeling 6(1), 83–110 (2007)
86. Moon, Y.-J.: Stochastic Models for Quality of Service of Component Connectors. PhD thesis, Leiden University (2011)
87. Moon, Y.-J., Silva, A., Krause, C., Arbab, F.: A compositional semantics for stochastic Reo connectors. In: Mousavi, Salaün (eds.) [88], pp. 93–107
88. Mousavi, M.R., Salaün, G. (eds.): Proceedings Ninth International Workshop on the Foundations of Coordination Languages and Software Architectures. EPTCS, vol. 30 (2010)
89. Mousavi, M.R., Sirjani, M., Arbab, F.: Formal semantics and analysis of component connectors in Reo. Electr. Notes Theor. Comput. Sci. 154(1), 83–99 (2006)
90. Odersky, M.: Report on the programming language Scala (2002), `http://lamp.epfl.ch/~odersky/scala/reference.ps`
91. Proença, J.: Synchronous Coordination of Distributed Components. PhD thesis, Leiden University (2011), `https://openaccess.leidenuniv.nl/handle/1887/17624`
92. Proença, J., Clarke, D.: Coordination models Orc and Reo compared. Electr. Notes Theor. Comput. Sci. 194(4), 57–76 (2008)
93. Proença, J., Clarke, D., de Vink, E.P., Arbab, F.: Decoupled execution of synchronous coordination models via behavioural automata. In: Mousavi, M.R., Ravara, A. (eds.) FOCLASA. EPTCS, vol. 58, pp. 65–79 (2011)
94. Rutten.: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. TCS: Theoretical Computer Science, 308 (2003)
95. Rutten, J.J.M.M.: Elements of stream calculus (an extensive exercise in coinduction). Electr. Notes Theor. Comput. Sci., 45 (2001)
96. Rutten, J.J.M.M.: A coinductive calculus of streams. Mathematical Structures in Computer Science 15(1), 93–147 (2005)
97. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York (2001)
98. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., van den Heuvel, W.-J.: Business Process Compliance Through Reusable Units of Compliant Processes. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 325–337. Springer, Heidelberg (2010)
99. Talcott, C.L., Sirjani, M., Ren, S.: Comparing three coordination models: Reo, ARC, and PBRD. Sci. Comput. Program. 76(1), 3–22 (2011)
100. Verhoef, C., Krause, C., Kanters, O., van der Mei, R.: Simulation-based performance analysis of channel-based coordination models. In: Meuter, Roman [82], pp. 187–201
101. Wegner, P.: Coordination as comstrainted interaction (extended abstract). In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 28–33. Springer, Heidelberg (1996)
102. Yokoo, M.: Distributed Constraint Satisfaction: Foundations of Cooperaton in Multi-Agent Systems. Springer Series on Agent Technology. Springer, New York (2000) NTT