Elements of Interaction

Farhad Arbab

Abstract. The most challenging aspect of concurrency involves the study of interaction and its properties. Interaction refers to what transpires among two or more active entities whose (communication) actions mutually affect each other. In spite of the long-standing recognition of the significance of interaction, classical 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, these formalisms turn interaction into 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.

Alternatively, we can view interaction as an explicit first-class concept, complete with its own composition operators that allow the specification of more complex interaction protocols by combining simpler, and eventually primitive, protocols. Reo [10, 11, 5] 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)

Farhad Arbab
Foundations of Software Engineering,
CWI
Science Park 123,
1098 XG Amsterdam,
The Netherlands
e-mail: farhad@cwi.nl

constructs that can be specified, verified, composed, and reused, independently of the actors that they may engage in disparate applications.

1 Introduction

Composition of systems out of autonomous subsystems pivots on coordination of concurrency. In spite of the fact that *interaction* constitutes the most challenging aspect of concurrency, contemporary models of concurrency predominantly treat interaction as a secondary or derived concept. Shared memory, message passing, calculi such as CSP [43], CCS [67], the π-calculus [68, 78], process algebras [29, 23, 40], and the actor model [7] represent popular approaches to tackle the complexities of constructing concurrent systems. Beneath their significant differences, all these models share one common characteristic: they are all *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 made available by the Green and the Red producers, alternating between the two.

Figure 1 shows the pseudo code for a typical 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. However, 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.

Global Names:
```
synchronization-points g, r, b, d
```

Green Producer:
```
G := genG(t) . ?g(k) . !b(t) . ?d(j) . !r(k) . G
```

Consumer:
```
B :=  ?b(t) . print(t) . !d("done") . B
```

Red Producer:
```
R := genR(t) . ?r(k) . !b(t) . ?d(j) . !g(k) . R
```

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

Fig. 2 Alternating producers and consumer in a process algebra

Process algebraic models fair only slightly better: they too embody an action-based model of concurrency. Figure 2 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 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 acknowledgement `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 1, while communication actions are concrete and explicit in the incarnation of our application in Figure 2, *interaction* is a manifestation of the model with no explicit structural correspondence.

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 the same synchronization point, the actions a_i and b_j "collide" with one another and their collision yields an interaction. Generally, the reason behind the specific collision of a_i and b_j remains debateable. Perhaps it was just dumb luck. Perhaps it was divine intervention. Some may prefer to attribute it to intelligent design! What is not debateable is the fact that, often, a split second earlier or later, perhaps in another run of the same application on a platform with a slightly different timing, a_i and b_j would collide not with each other, but with two other actions (of perhaps other processes) yielding completely different interactions. 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 parts in a protocol, and as an ephemeral manifestation, a protocol itself becomes 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 reusable, tangible protocols that can be constructed and verified independently of the processes that they engage.

2 Interaction Centric Concurrency

The fact that we currently use languages and tools based on various concurrent object oriented models, actor models, various process algebras, etc., simply means that these models comprise the best in our available arsenal to tackle the complexity of concurrent systems. However, this fact does not mean that these languages and tools necessarily embody the most appropriate models for doing so. We observe that action-centric models of concurrency

turn interaction into an implicit by-product of the execution of actions. We also observe that the most challenging aspect of concurrent systems is their interaction protocols, whose specification and study can become simpler in a model where interaction is treated as a first-class concept[1]. These observations serve as motivation to consider an interaction-centric model of concurrency, instead.

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 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 [79]. 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 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* as 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, and 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.

[1] A notion constitutes a first-class concept in a model only if the model provides structural primitives to directly define instances of it, together with operators to compose and manipulate such instances by composing and manipulating their respective structures. Thus, "process" constitutes a first-class concept in process algebras, but "interaction/protocol" does not.

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 component, service, actor, etc.) offers no methods, functions, or procedures for other entities to call, and it makes no such calls itself. Moreover, processes cannot exchange message 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 performing *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 uni-directional, 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. 3 Protocol in a connector

Inter-process communication is possible only by mediation of connectors. For instance, Figure 3 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 Figure 1: it contains no semaphore operations or any other inter-process communication primitives.

The direction of the connector arrow in Figure 3 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 repeat, 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 3 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"[2]. In contrast, the interaction protocol and coordination in the examples in Figures 1 and

[2] 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.

2 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 3 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 3, 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 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 3 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 [10, 11, 5] is a channel-based exogenous coordination model 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 un-targeted 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 interconnects.

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 4 shows a common set of primitive channels often used to build Reo connectors.

Fig. 4 A typical set of Reo channels

A *Sync*, for short, has a source and a sink end and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink.

A *LossySync*, for short, is similar to synchronous channel except that it always accepts all data items through its source end. The data item is transferred if it is possible for the data item to be dispensed through the sink end, otherwise the data item is lost.

A *FIFO1* channel represents an asynchronous channel with one buffer cell which is empty if no data item is shown in the box (this is the case in Figure 1). If a data element d is contained in the buffer of a FIFO1 channel then d is shown inside the box in its graphical representation.

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 since 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, but never simultaneously.

For a *filter channel*, or Filter(P), its pattern $P \subseteq Data$ specifies the type of data items that can be transmitted through the channel. This channel accepts a value $d \in P$ 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 $d \notin P$ through the source end and loses them immediately.

Synchronous and asynchronous *Spouts* each is a dual of its respective drain channel, as they have two sink ends through which they produce nondeterministic data items. Further discussion of these and other primitive channels is beyond the scope of this paper.

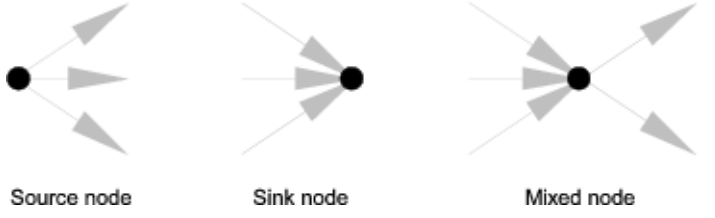

Fig. 5 Reo nodes

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 5 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 component connector. The hidden nodes can no longer be accessed or observed from outside.

The term *boundary nodes* is also sometimes used to collectively refer to source and sink nodes. Boundary nodes define the interface of a connector. Components connect to the boundary nodes of a connector and interact anonymously with each other through this interface by performing I/O operations on the boundary nodes of the connector: *take* operations on sink nodes, and *write* operations on source nodes.[3]

At most one component can be connected to a (source or sink) node at a time. The I/O operations are performed through interface nodes of components which are called ports. We identify each node with a name, taken from a set $Names$, with typical members $A, B, C, \ldots$. For an arbitrary node A, we use d_A as the symbol for the observed data item at A.

[3] The `get` and `put` operations mentioned in the description of the components in Figure 3 are higher-level wrappers around the primitive *take* and *write* operations of Reo.

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. 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. This is not a trivial task. In 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 [36] for a detailed discussion of this.

Reo has been used for composition of Web services [53, 62, 19], modeling and analysis of long-running transactions and compliance in service-oriented systems [59, 18, 58], coordination of multi agent systems [12], performance analysis of coordinated compositions [21], and modeling of coordination in biological systems [35].

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 and more. 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 be then 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 it suitable for exogenous coordination (which, in fact, makes it simpler). Similar to the producer `P` in Figure 3, this code now consists of an infinite loop, in each

iteration of which a 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 3.

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 these revised versions of producers and consumer processes. These examples serve three purposes. First, they show a flavor of programming pure interaction coordination protocols as Reo circuits. Second, they present a number of generically useful circuits that can be used 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.

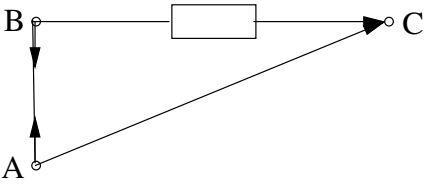

Fig. 6 Reo circuit for Alternator

4.1 Alternator

The connector shown in Figure 6 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. The empty buffer 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.

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 to node A, the output port of the revised Red producer to node B, and the input port of the revised consumer to node C of the Reo circuit in Figure 6. 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 1 and 2. As explained above, the Reo circuit in Figure 6 requires the availability of a pair of values (from the Green producer) on A and (from the Red producer on) B 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 1

and 2 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 6 is *not* what we want, we need to construct a different Reo circuit to impose the same behavior as in Figures 1 and 2. This is precisely what we describe below.

4.2 Sequencer

Figure 7(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.

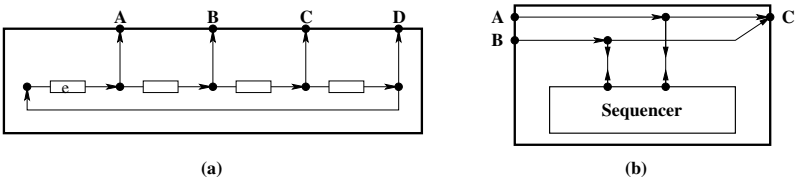

Fig. 7 Sequencer

Figure 7(b) shows a simple example of the utility of the sequencer. The connector in this figure consists of a two-node sequencer, plus a pair of *Sync* channels and a *SyncDrain* 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 6, 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 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 6 and 7(b). The alternator in Figure 6 delays the transfer of a data item from A to C until a data item is also available at B. The circuit in Figure 7(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 to node A, the output port of the Red producer to node B, and the input port of the consumer to node C. The behavior of this version is now the same as those in Figures 1 and 2. The circuit in Figure 7(b) embodies the same protocol that is implicit in Figures 1 and 2.

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.

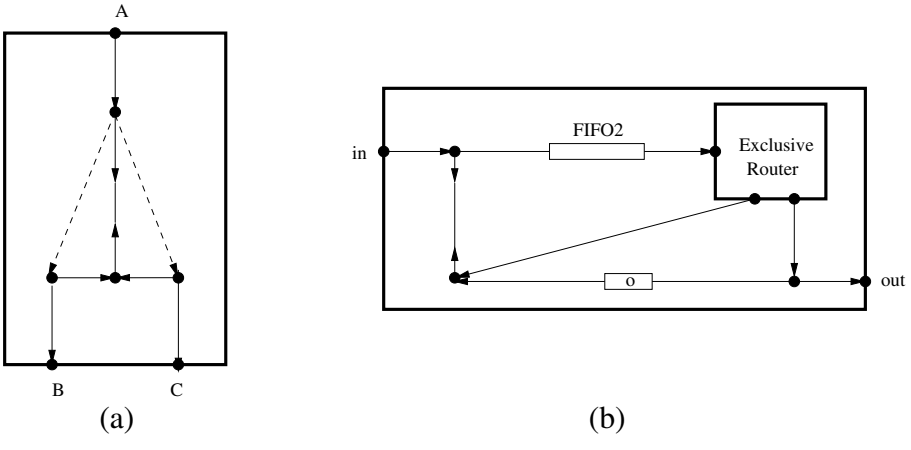

Fig. 8 An exclusive router and a shift-lossy `FIFO1`

4.3 Exclusive Router

The connector shown in Figure 8(a) is an *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 nodes B and 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 lower-middle mixed node: 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*.

4.4 Shift-Lossy FIFO1

Figure 8(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`). 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. The connector in Figure 8(b) is composed of an exclusive router (shown in Figure 8(a)), an initially full `FIFO1` channel, an initially empty `FIFO2` channel, and four `Sync` channels. The `FIFO2` channel itself is composed out of two sequential `FIFO1` channels. See [27] for a more formal treatment of the semantics of this connector.

The shift-lossy `FIFO1` circuit in Figure 8(b) is indeed so frequently useful as a connector in construction of more complex circuits, that it makes sense to have a special graphical symbol to designate it as a short-hand. Figure 9 shows a circuit that uses two instances of our shift-lossy `FIFO1`. The graphical symbol we use to represent a shift-lossy `FIFO1` "channel" is intentionally similar to 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 Dataflow Variable

The Reo circuit in Figure 9 implements the behavior of a dataflow variable. It uses two instances of the shift-lossy `FIFO1` connector shown Figure 8(b), to build a connector with a single input and a single output nodes. Initially, the buffers of its shift-lossy `FIFO1` channels are empty, so an initial take on its output node suspends for data. Regardless of the status of its buffers, or whether or not data can be dispensed through its output node, every write to its input node always succeeds and resets both of its buffers to contain the new data item. Every time a value is dispensed through its output node, a copy of this value is "cycled back" into its left shift-lossy `FIFO1` 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.

The variable circuit in Figure 9 is also very frequently useful as a connector in construction of more complex circuits. Therefore, it makes sense to have a short-hand graphical symbol to designate it with as well. Figure 10 shows 3

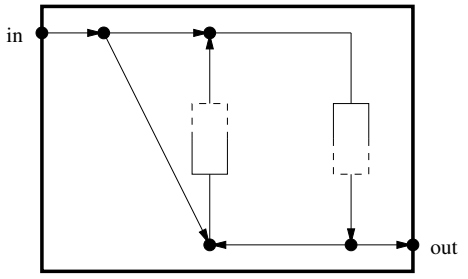

Fig. 9 Dataflow variable

instances of our variable used in two connectors. Our symbol for a variable is similar to that for a regular `FIFO1` channel, except that we use a rounded box to represent its buffer: the rounded box hints at the recycling behavior of the variable circuit, which implements its remembering of the last data item that it obtained or dispensed.

4.6 Decoupled Alternating Producers and Consumer

Figure 10(a) shows how the variable circuit of Figure 9 can be used to construct a version of the example in Figure 3, 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.

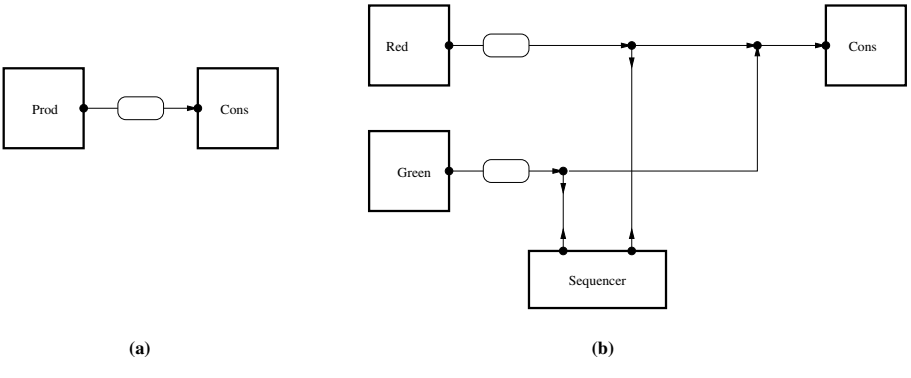

Fig. 10 Decoupled producers and consumer

The connector in Figure 10(b) is a small variation of the Reo circuit in Figure 7(b), with two instances of the variable circuit of Figure 9 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, 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.

4.7 Flexibility

Figures 6, 7(b), and 10(b) show three 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 1 and 2, which correspond to the protocol of the circuit in Figure 7(b), in order to achieve the behavior induced by the circuit in Figure 6 or 10(b).

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 [2].

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, 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 [76, 75, 77]. 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* [22]. 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	$\langle \alpha, a\rangle$ `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)	$\langle \alpha, a\rangle$ `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. 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 4, as well as that of the merge and replication behavior inherent in Reo nodes. 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 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 8(a).

5.2 Constraint Automata

Constraint automata provide an operational model for the semantics of Reo circuits [27]. 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). 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.

Constraint automata have been used for the verification of their properties through model-checking [6, 50, 30, 26]. Results on equivalence and containment of the languages of constraint automata [27] provide opportunities for analysis and optimization of Reo circuits.

The constraint automata semantics of Reo is used to generate executable code for Reo [17]. 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.

Variants of the constraint automata model have been devised to capture time-sensitive behavior [13, 47, 48], probabilistic behavior [24], stochastic behavior [28], context sensitive behavior [31, 38, 45], resource sensitivity [64], and QoS aspects [65, 15, 16, 70] 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 [36]. 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. A more sophisticated model using three colors is necessary to capture the context sensitive behavior of primitives such as the *LossySync* channel. A proof-of-concept for this semantic model was built to show how it can coordinate Java threads as components.

The CC model is also used in the implementation of a visualization tool that produces Flash animations depicting the behavior of a connector [38, 73].

Finding a consistent coloring for a circuit amounts to constraint satisfaction. Constraint solving techniques [9, 80] have been applied using the CC model to search for a valid global behavior of a given Reo connector [37]. 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 [1, 73] 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 [71] 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 [41] semantics of Reo offers a uniform setting for representing not only the ordinary dataflow execution of Reo connectors, but also their dynamic reconfigurations [14]. An abstraction of the constraint automata is used in [61] to serves 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 [33], which also shows the close semantic link between Reo and zero-safe variant of Petri nets. A comparison of Orc [69, 49] and Reo appears in [74].

The semantics of Reo has also been formalized in the Unifying Theories of Programming (UTP) [44]. The UTP approach provides a family of algebraic operators that interpret the composition of Reo connectors more explicitly than in other approaches [66]. 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 [8].

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 [34]. Graph transformations techniques have been used in combination with the connector coloring model to formalize the dynamic reconfiguration semantics of Reo circuits triggered by dataflow [52, 51].

6 Tools

Tool support for Reo consists of a set of Eclipse plug-ins that together comprise the Eclipse Coordination Tools (ECT) visual programming environment [2]. 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 plugins. The animation plugin 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 [38, 73].

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 plugin 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 [6, 30, 50, 25, 26]. 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 [13]. A SAT-based bounded model checker exists for verification of a variant of TCA [47, 48], 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.

Another means for verification of Reo is made possible by a transformation bridge into the mCRL2 toolset [3, 42]. 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 [55, 56, 57]. 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.

A CA code generator plugin 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 [54, 63].

A distributed implementation of Reo exists [1] as a middleware in the actor-based language Scala [72], 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 [73].

A set of ECT plugin tools are under development to support coordination and composition of Web Services using Reo. ECT plugins are available for automatic conversion of coordination and concurrency models expressed as UML sequence diagrams [19, 20], BPMN diagrams [18], and BPEL source code into Reo circuits [32].

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 [15, 16, 70]. A bridge to Prism [4] allows further analysis of the resulting Markov chains [21]. 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 [46].

Based on algebraic graph transformations, a reconfiguration engine is available as an ECT plugin that supports dynamic reconfiguration of distributed Reo circuits triggered by dataflow [17, 51]. 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 plugins 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.

References

1. Distributed Reo, http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Redrum/BigPicture
2. Eclipse coordination tools home page, http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools
3. mcrl2 home page, http://www.mcrl2.org
4. Prism, http://www.prismmodelchecker.org
5. Reo home page, http://reo.project.cwi.nl
6. Vereofy home page, http://www.vereofy.de/
7. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
8. 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, Los Alamitos (2009)
9. Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)

10. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14(3), 329–366 (2004)
11. Arbab, F.: Abstract Behavior Types: a foundation model for components and their composition. Sci. Comput. Program. 55(1-3), 3–52 (2005)
12. Arbab, F., Astefanoaei, L., de Boer, F.S., Dastani, M., Meyer, J.-J.C., Tinnemeier, N.A.M.: 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)
13. 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)
14. 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)
15. 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)
16. 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 [39], pp. 268–287
17. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y.-J., Proença, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In: Tool demo session at FACS 2008 (2008)
18. Arbab, F., Kokash, N., Meng, S.: Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA. Communications in Computer and Information Science, vol. 17, pp. 108–123. Springer, Heidelberg (2008)
19. Arbab, F., Meng, S.: Synthesis of connectors from scenario-based interaction specifications. In: Chaudron, M.R.V., Szyperski, C., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 114–129. Springer, Heidelberg (2008)
20. Arbab, F., Meng, S., Baier, C.: Synthesis of Reo circuits from scenario-based specifications. Electr. Notes Theor. Comput. Sci. 229(2), 21–41 (2009)
21. 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, New York (2009)
22. 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)
23. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, Cambridge (1990)
24. Baier, C.: Probabilistic models for Reo connector circuits. Journal of Universal Computer Science 11(10), 1718–1748 (2005)
25. 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)
26. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field, Vasconcelos [39], pp. 247–267
27. 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)

28. 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)
29. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60, 109–137 (1984)
30. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. Electr. Notes Theor. Comput. Sci. 215, 209–226 (2008)
31. Bonsangue, M.M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Field, Vasconcelos [39], pp. 184–203
32. 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
33. 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)
34. Clarke, D.: A basic logic for reasoning about connector reconfiguration. Fundam. Inform. 82(4), 361–390 (2008)
35. 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)
36. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Sci. Comput. Program. 66(3), 205–225 (2007)
37. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Deconstructing Reo. Electr. Notes Theor. Comput. Sci. 229(2), 43–58 (2009)
38. Costa, D.: Formal Models for Context Dependent Connectors for Distributed Software Components and Services. Leiden University (2010)
39. Field, J., Vasconcelos, V.T. (eds.): COORDINATION 2009. LNCS, vol. 5521. Springer, Heidelberg (2009)
40. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer, Heidelberg (1999)
41. 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)
42. 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, Germany (2006)
43. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
44. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall, London (1998)
45. 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, Los Alamitos (2008)
46. Kanters, O.: QoS analysis by simulation in Reo. Vrije Universiteit Amsterdam (2010)
47. Kemper, S.: SAT-based Verification for Timed Component Connectors. Electr. Notes Theor. Comput. Sci. 255, 103–118 (2009)

48. 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)
49. 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)
50. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Electr. Notes Theor. Comput. Sci 175(2), 19–37 (2007)
51. 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)
52. 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); ECEASST, ISSN 1863-2122, http://www.easst.org/eceasst/
53. Koehler, C., Lazovik, A., Arbab, F.: Reoservice: Coordination modeling tool. In: Krämer et al [60], pp. 625–626
54. 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)
55. 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)
56. Kokash, N., Krause, C., de Vink, E.P.: Time and data-aware analysis of graphical service models in Reo. In: SEFM 2010: Proc. 8th IEEE International Conference on Software Engineering and Formal Methods. IEEE, Los Alamitos (to appear, 2010)
57. Kokash, N., Krause, C., de Vink, E.P.: Verification of context-dependent channel-based service models. In: FMCO 2009: Formal Methods for Components and Objects: 8th International Symposium. LNCS. Springer, Heidelberg (to appear, 2010)
58. 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)
59. 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, New York (2009)
60. Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.): ICSOC 2007. LNCS, vol. 4749. Springer, Heidelberg (2007)
61. 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)
62. Lazovik, A., Arbab, F.: Using Reo for service coordination. In: Krämer et al. [60], pp. 398–403
63. Maraikar, Z., Lazovik, A.: Reforming mashups. In: Proceedings of the 3rd European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2008), June 2008. Imperial College, London (2008)

64. 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)
65. Meng, S., Arbab, F.: QoS-driven service selection and composition. In: Billington, J., Duan, Z., Koutny, M. (eds.) ACSD, pp. 160–169. IEEE, Los Alamitos (2008)
66. Meng, S., Arbab, F.: Connectors as designs. Electr. Notes Theor. Comput. Sci. 255, 119–135 (2009)
67. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
68. Milner, R.: Elements of interaction - turing award lecture. Commun. ACM 36(1), 78–89 (1993)
69. Misra, J., Cook, W.R.: Computation orchestration. Software and System Modeling 6(1), 83–110 (2007)
70. Moon, Y.-J., Silva, A., Krause, C., Arbab, F.: A compositional semantics for stochastic Reo connectors. In: Proceedings of the 9th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA (2010)
71. 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)
72. Odersky, M.: Report on the programming language Scala (2002), http://lamp.epfl.ch/~odersky/scala/reference.ps
73. Proença, J.: Dreams: A Distributed Framework for Synchronous Coordination. Leiden University (2011)
74. Proença, J., Clarke, D.: Coordination models orc and reo compared. Electr. Notes Theor. Comput. Sci. 194(4), 57–76 (2008)
75. Rutten: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. TCS: Theoretical Computer Science 308 (2003)
76. Rutten, J.J.M.M.: Elements of stream calculus (an extensive exercise in coinduction). Electr. Notes Theor. Comput. Sci. 45 (2001)
77. Rutten, J.J.M.M.: A coinductive calculus of streams. Mathematical Structures in Computer Science 15(1), 93–147 (2005)
78. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York (2001)
79. 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)
80. Yokoo, M.: Distributed Constraint Satisfaction: Foundations of Cooperaton in Multi-Agent Systems. Springer Series on Agent Technology. Springer, New York (2000); NTT