Coordinating Multicore Computing

Farhad Arbab[1,2]([✉]) and Sung-Shik T.Q. Jongmans[1]

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

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, 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 [4,7,8,15] serves as a premier example of such an interaction-based model of concurrency. In this paper, we describe Reo and its compiler. 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.

This paper complements the first author's lecture at the 15th *International School on Formal Methods for the Design of Computer, Communication and Software Systems* in Bertinoro, Italy, June 2015, and collects previously published material (notably [9]).

1 Introduction

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. Concurrency is inherently difficult because it involves complex interaction protocols. 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.

Global Objects:

```
 1 Semaphore greenSemaphore = new Semaphore(1);
 2 Semaphore redSemaphore = new Semaphore(0);
 3 Semaphore bufferSemaphore = new Semaphore(1);
 4 String buffer = EMPTY;
```

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 }
```

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 }
```

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

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 [40], CCS [68], the π-calculus [69,72], process algebras [19,27,36], and the actor model [6] 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.

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 livelock, 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

```
Green Producer:                    Red Producer:
14 while (true) {                  28 while (true) {
15   sleep(5000);                  29   sleep(3000);
16   greenText = ...;              30   redText = ...;
17   greenSemaphore.acquire();     31   redSemaphore.acquire();
18   while (greenText !=EMPTY) {   32   while (redText !=EMPTY) {
19     bufferSemaphore.acquire();  33     bufferSemaphore.acquire();
20     if (buffer == EMPTY) {      34     if (buffer == EMPTY) {
21       buffer = greenText;       35       buffer = redText;
22       greenText = EMPTY;        36       redText = EMPTY;
23     }                           37     }
24     bufferSemaphore.release();  38     bufferSemaphore.release();
25   }                             39   }
26   redSemaphore.release();       40   greenSemaphore.release();
27 }                               41 }
```

Fig. 2. Busy waiting consumer

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 Fig. 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 to say that we are sure all our 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 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 Fig. 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 Fig. 2. A disadvantage of this code is that it more heavily uses the busy-waiting mechanism that already existed in the consumer code in Fig. 1. A better alternative is to use a different protocol that explicitly respects the turn taking, as described below.

In the program shown in Fig. 3, the consumer too has its own turn-taking semaphore, the new `blueSemaphore` (line 3), which is initialized to be locked,

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

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 }
```

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

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

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 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

```
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

parameterize it such that we can instantiate the protocol with arbitrary numbers
of processes containing arbitrary computation code, 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 of participants and the specific compu-
tation code for each, to instantiate 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 order-
ing 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 diffi-
cult with this and similar incarnations of a protocol because they require *seeing*
and *touching* the protocol as a tangible concrete entity.

Process algebraic models of concurrency fare 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 alter-
nating 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 con-
sidered 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. Typ-
ical composition operators include sequential composition _ . _, parallel composi-
tion _ | _, 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 Fig. 3, while communication actions are concrete and explicit in the incarnation of our application in Fig. 4, *interaction* is a manifestation of the model with no direct explicit structural correspondence. 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. Nevertheless, process algebras and calculi also constitute action-based models of concurrency.

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 ith communication action a_i on a synchronization point, at the same time that another process B happens to execute its jth 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 [74]. 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

Fig. 5. Protocol in a connector

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`.

Inter-process communication is possible only by mediation of connectors. For instance, Fig. 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 Figs. 1, 2, and 3: it contains no semaphore operations or any other inter-process communication primitives.

The direction of the connector arrow in Fig. 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, Fig. 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 (or processes) themselves, and without their "knowledge"[1]. In contrast, the interaction protocol and coordination in the examples in Figs. 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 Fig. 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 Fig. 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.

Nevertheless, 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. Encoding every instance of these recurring concepts in terms of assignment statements, if-then-else, for-loops, and communication actions in every application is tedious, error prone, and obscures the concepts beyond recognition when they are interspersed with the computation code of an application. Compliant with the constraint view of interaction advocated above, these concepts can be expressed more succinctly and elegantly as constraints. 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

---

[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.

protocols and the connectors that embody them, of which the connector in Fig. 5 is but a trivial example. Reo, described in the next section, is a premier example of such a language.

3   Overview of Reo

Reo [4,7,8,15] is a channel-based exogenous coordination language wherein complex coordinators, called connectors, or *circuits*, 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 a semantic model of Reo) 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.
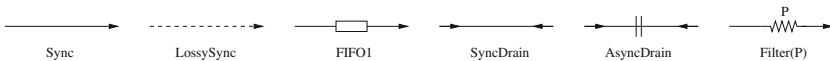

Fig. 6. A typical set of Reo channels

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.

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 Fig. 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 $P \subseteq Data$ 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 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.

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
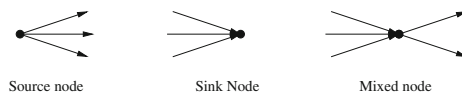

Source node   Sink Node   Mixed node

Fig. 7. Reo nodes

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 node 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 process 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 processes in Fig. 5 trigger *write* and *take* operations of Reo on the channel ends of their respective nodes.

A component (or process) 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 (or process) 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 [33]. 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 [32] for a detailed discussion of this.

Reo has been used for composition of Web services [16,57,65], modeling and analysis of long-running transactions in service-oriented systems [60], coordination of multi-agent systems [10], performance analysis of coordinated compositions [12,13,17,70,71], modeling of business processes and verification of their compliance [14,59,73], and modeling of coordination in biological systems [31].

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.

Tool support for Reo consists of a set of Eclipse plug-ins that together comprise the Extensible Coordination Tools (ECT) visual programming environment [2]. The Reo graphical editor supports drag-and-drop graphical composition and editing of Reo connectors. 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 connector, which provides an intuitive insight into their behavior through visualization of how they work. Several model checking tools are available for analyzing Reo. The Vereofy model checker, integrated in ECT, is based on constraint automata [5,21–24,28,37,55,56]. Properties of Reo connectors 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). Another means for verification of Reo is made possible by a transformation bridge into the mCRL2 toolset [3,38]. 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 [58,62,63]. Through mCRL2, it is possible to verify the behavior of timed Reo connectors, or Reo connectors 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]. Another tool is a Reo compiler that generates executable code for Reo connectors; we discuss compilation in more detail shortly. Even more tools are discussed elsewhere [9].

4   Examples

Recall our alternating producers and consumer example of Sect. 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 Fig. 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

```
Consumer:                    Green Producer:                    Red Producer:
  1 while (true) {             6 while (true) {                   11 while (true) {
  2   sleep(4000);             7   sleep(5000);                   12   sleep(3000);
  3   get(input, text);        8   greenText = ...;               13   redText = ...;
  4   print(text);             9   put(output, greenText);        14   put(output, redText);
  5 }                         10 }                                15 }
```

Fig. 8. Generic reusable producers and consumer

revise the consumer code, fashioning it after the consumer C in Fig. 5. Figure 8 shows this code.

In the remainder of this section, we present a number of protocols to implement different versions of the alternating producers and consumer example of Sect. 1, using the producers and consumer processes in Fig. 8. These examples serve three purposes. First, they show a flavor of programming of pure interaction coordination protocols as Reo connectors. Second, they present a number of generically useful connectors that can serve as connectors in many other applications, or as sub-connectors in the connectors 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.

4.1   Alternator

The connector shown in Fig. 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 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.
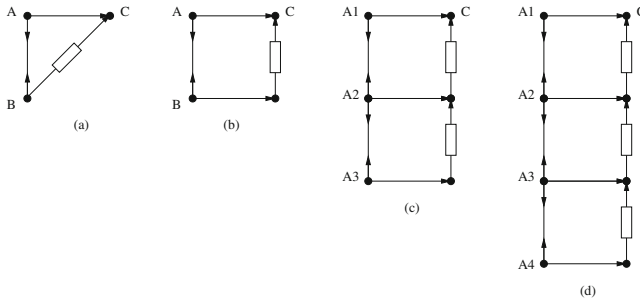

Fig. 9. Reo connectors for alternators

The connector in Fig. 9(b) has an extra `Sync` channel between node B and the `FIFO1` channel, compared to the one in Fig. 9(a). It is trivial to see that these two connectors have the exact same behavior. However, the structure of the connector in Fig. 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 Fig. 9(c) and (d).

The two `SyncDrain` channels in the connector shown in Fig. 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 Fig. 9(d) is obtained by replicating the one in Fig. 9(b) 3 times. Following the reasoning for the connector in Fig. 9(c), it is easy to see that the connector in Fig. 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 Sect. 1 can now be composed by attaching the output port of the revised Green producer in Fig. 8 to node A, the output port of the revised Red producer in Fig. 8 to node B, and the input port of the consumer in Fig. 8 to node C of the Reo connector in Fig. 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 Figs. 3 and 4. As explained above, the Reo connector in Fig. 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 Figs. 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 Sect. 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 Fig. 9(a) is *not* what we want, we need to construct a different Reo connector to impose the same behavior as in Figs. 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
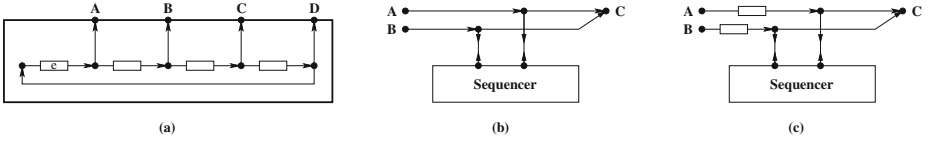
Fig. 10. Sequencer

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 connector in Fig. 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 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 connectors in Figs. 9(a) and 10(b). The alternator in Fig. 9(a) delays the transfer of a data item from A to C until a data item is also available at B. The connector in Fig. 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 Fig. 8 to node A, the output port of the Red producer in Fig. 8 to node B, and the input port of the consumer in Fig. 8 to node C. The behavior of this version of our application is now the same as the programs in Fig. 4 and in Fig. 1 (after replacing its producers with the ones in Fig. 2). The connector in Fig. 10(b) embodies the same protocol that is implicit in Fig. 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 Sect. 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 connector for such a protocol.

4.3   Buffered Sequencing

Figure 10(c) shows how easily we can decouple the producers from the consumer by adding two `FIFO1` channels to the connector in Fig. 10(b). The protocol implemented by this connector allows each producer to move ahead of its turn by one item. Obviously, one can add more `FIFO1` channels, as desired, to allow the producers to move ahead of their turns by any arbitrary k items, before they need to wait for their next output item to be accepted. Because Reo allows users to define arbitrary channels, it is equally possible to define an unbounded `FIFO` channel, and use two instances of this channel to allow producers to move ahead of the consumer by any arbitrary number of items.

A characteristic of all such buffered protocols is that they make sure every item produced by every producer is eventually consumed by the consumer. In fact, such total retention of data is not always desirable. Sometimes, some sort of *sampling* is required to ensure the consumer is not overwhelmed by much faster producers, or to ensure that the consumer always processes the most up-to-date produced items.

4.4   Sampling

The connector in Fig. 11(a) is a variant of the one in Fig. 10(b) which never delays any of its producers. Producers can produce items as fast as they wish and the protocol never delays them; it simply loses any item that they produce when the consumer is not ready to take it. Whenever the consumer is ready to take an item, it must wait for the producer whose turn it is to produce its next item for it to consume. On the one hand, this ensures that the consumer always obtains the freshest, most up-to-date item produced by each producer. On the other hand, although the producers never wait, the consumer may still have to wait for the right producer to deliver its next fresh item. If this is not desirable, we may wish the protocol to hold at least one produced item at hand to alleviate the need for the consumer to wait.

The connector on the left-hand side of the $\equiv$ sign in Fig. 11(b) shows a useful connector which behaves almost exactly as a `FIFO1` channel. The only difference is that, unlike a normal `FIFO1` channel, this connector does not suspend its writer if its buffer is full; it allows the write to succeed, but loses the written data. We use the symbol on the right-hand side of the $\equiv$ sign in Fig. 11(b) as a short-hand for this connector, and refer to it as an `OverflowLossyFIFO1` channel. This symbol is intentionally similar to that of a regular `FIFO1` channel, because the behavior
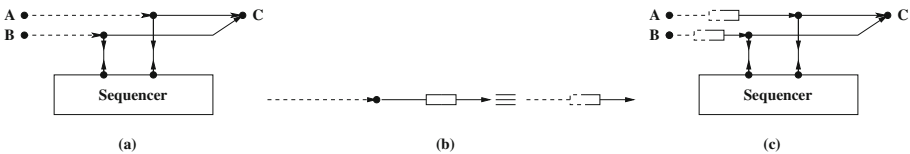

Fig. 11. Synchronized sampling, `OverflowLossyFIFO1`, and buffered sampling

of this connector closely resembles that of a regular `FIFO1` channel. The dashed source-side half of this channel suggests that when its buffer is full, this channel simply loses its new input items, as if they "overflow" over a full container.

Replacing the `FIFO1` channels in Fig. 10(c) with such `OverflowLossyFIFO1` channels, we obtain the connector in Fig. 11(c). Using this connector in our running example application allows the producers to run as fast as they wish, and allows the consumer to merely sample what each producer delivers. If the consumer ever gets ahead of a producer by more than one cycle, then this protocol makes the consumer wait. Obviously, we can add more `FIFO1` channels to the construct in Fig. 11(b) to obtain an `OverflowLossyFIFO`k channel, for any $k > 1$. We can then raise the sampling depth of our protocol to any k by using `OverflowLossyFIFO`k channels in connectors similar to the one in Fig. 11(c). Such a protocol with the sampling depth of k allows the consumer move ahead of a producer by k items, while the protocol retains up to k items produced by each producer, before it loses their excess output.

A consequence of using `OverflowLossyFIFO`k channels in the above connectors is that the protocol tends to retain the "oldest" k sampled output of each producer.[2] In many situations, it is desirable to bias sampling toward most recent values, discarding older values. To do this, we need a counterpart of the `OverflowLossyFIFO1` channel in Fig. 11(b), that when its buffer is full, discards the old value in the buffer and retains its new input. We present a connector with such behavior in Sect. 4.6.

4.5   Exclusive Router

The connector shown in Fig. 12(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

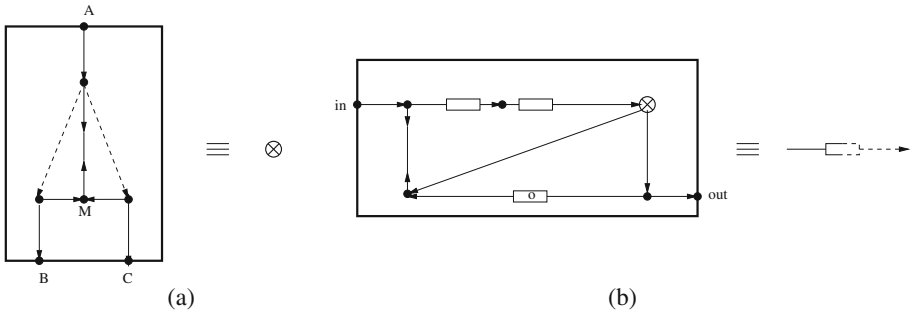(a)   (b)

Fig. 12. An exclusive router and a `ShiftLossyFIFO1`

---

[2] In fact, this characterization is not very accurate for values of $k > 1$. Work out what happens for $k = 2$, for instance.

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 connectors. The crossed circle shown on the right-hand side of the $\equiv$ symbol in Fig. 12(a) is the symbol that we use to represent a generic n-ary exclusive router.

4.6   Shift-Lossy FIFO1

Figure 12(b) shows a Reo connector 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 Fig. 12(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 [25] for a more formal treatment of the semantics of this connector.

The ShiftLossyFIFO1 connector in Fig. 12(b) is indeed so frequently useful as a connector in construction of more complex connectors, 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 the $\equiv$ symbol in Fig. 12(b) is the what we use to represent this connector, 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 connector closely resembles that of a regular FIFO1 channel. The dashed sink-side half of this channel suggests that it loses the older values to make room for new arrivals, i.e., it shifts to lose.

4.7   Decoupled Alternating Producers and Consumer

Figure 13(a) shows how the ShiftLossyFIFO1 connector of Fig. 12(b) can be used to construct a version of the example in Fig. 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
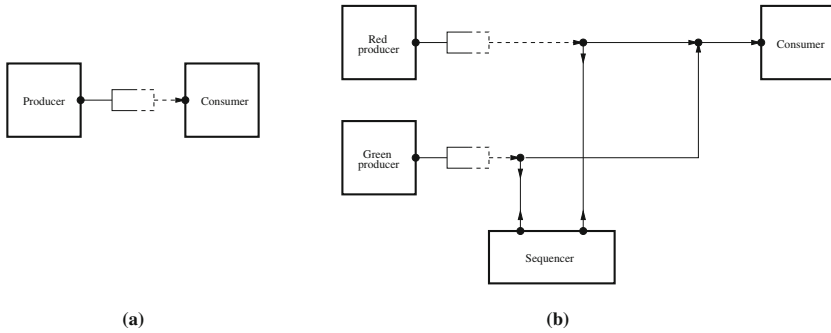
Fig. 13. Decoupled producers and consumer

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.

The connector in Fig. 13(b) is a small variation of the Reo connector in Fig. 10(b), with two instances of the `ShiftLossyFIFO1` connector of Fig. 12(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.

4.8   Dataflow Variable

The Reo connector in Fig. 14 implements the behavior of a dataflow variable. It uses two instances of the `ShiftLossyFIFO1` connector shown Fig. 12(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 connector "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.
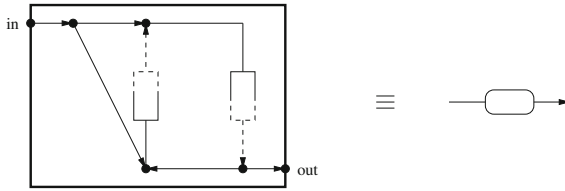
Fig. 14. Dataflow variable

The variable connector in Fig. 14 is also very frequently useful as a connector in construction of more complex connectors. Therefore, it makes sense to have a short-hand graphical symbol to designate it with as well. The symbol shown on the right-hand side of Fig. 14 is the what we use to represent this connector, 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 connector 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 connector, which implements its remembering of the last data item that it obtained or dispensed.

4.9   Fully Decoupled Alternating Producers and Consumer

Figure 15(a) shows how the variable connector of Fig. 14 can be used to construct a version of the example in Fig. 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.
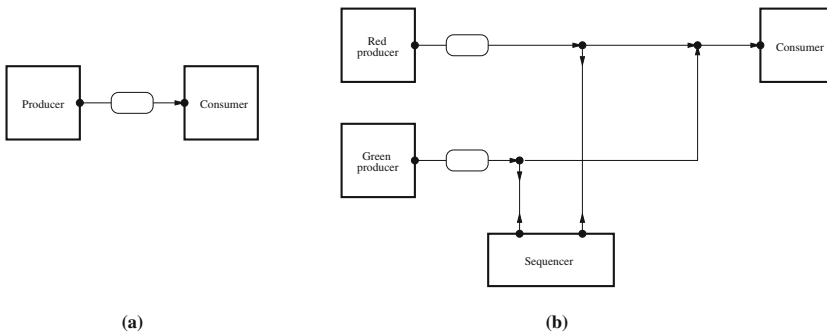

Fig. 15. Fully decoupled producers and consumer

The connector in Fig. 15(b) is a small variation of the Reo connector in Fig. 10(b), with two instances of the variable connector of Fig. 14 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.

4.10 Flexibility and Scaling

Figures 9(a), 10(b), (c), 11(a), (c), 13(b), and 15(b) show a number of different connectors, each imposing a variant of a protocol for the coordination of two alternating producers and a consumer. The exact same producers and consumer processes can be combined with any of these connectors 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 Figs. 3 and 4, which correspond to the protocol of the connector in Fig. 10(b), in order to achieve the behavior induced by the connector in Figs. 9(a), 10(c), 11(a), (c), 13(b), or 15(b). It is also instructive to compare the ease with which any of these connectors can be parameterized to scale up their number of producers, with the changes necessary to scale up the number of producers in action-centric versions of these protocols.

Moreover, applications with many producers may indeed require somewhat different treatment of the output of some of their producers. For instance, an application may require barrier synchronization of some producers, synchronous sampling of some others, buffered sampling of yet others, etc., etc. It is trivial to mix-and-match the necessary interaction (sub-)protocols that we examined, to tailor make such a protocol, essentially through cut-and-paste of parts of the various Reo connectors presented above. Such cut-and-paste is generally unthinkable when protocols are expressed in terms of action-based constructs of traditional models of concurrency.

As if anyone needed more evidence to appreciate that concurrency is difficult, the many variants of our deceptively trivial running example presented above, plus the multitudes of their possible mix-and-match variants, demonstrate that even seemingly trivial protocols involve intricate details that require careful attention and explicit, concrete, first-class treatment. By the way, none of the variants of the Reo connectors presented above captures the behavior of the Java-like code of our initial attempt. For the sake of completeness, the behavior of the protocol in Fig. 1 corresponds to the behavior of the connector in Fig. 16. Just as in the case of the program in Fig. 1, this connector allows the producers at nodes A and B alternate and over-write each other in the buffer of the `ShiftLossyFIFO1`. The consumer at C can obtain only the latest value produced by either of the producers.
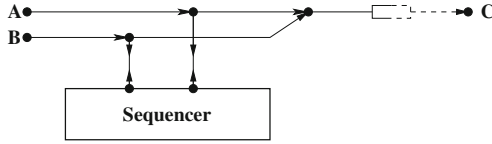
Fig. 16. Alternating and over-writing

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 [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, 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 constraint automata [25], the main semantics used in verification and code generation; a comprehensive overview of other models appears elsewhere [44].

Constraint automata provide an operational model for the semantics of Reo connectors. The states of an automaton represent the configurations of its corresponding connector (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, Fig. 17 shows the constraint automata semantics for some of the common Reo primitives.

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
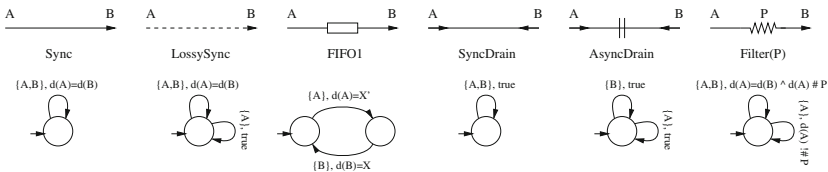

Fig. 17. Constraint automata of some typical Reo channels

$d(A) = d(B)$ 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 Sect. 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.

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.

---

[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 [29, 34]. 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 [52, 61].
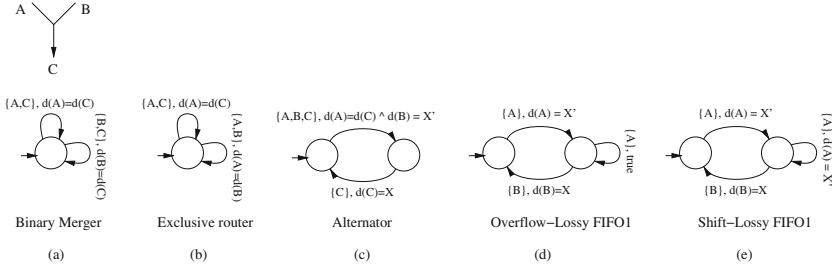
Fig. 18. Constraint automata of a binary merger and some example connectors

The semantics of a Reo connector 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 [25]. 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 18(a) shows the constraint automaton for a nondeterministic binary merge primitive.

Figure 18(b) shows the constraint automaton representing the semantics of the exclusive router Reo connector of Fig. 12(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 18(c) shows the constraint automaton representing the semantics of the alternator connector of Fig. 9(a), obtained as the product of the constraint automata of its constituent `Sync` channel, `SyncDrain` channel, `FIFO1` channel, and merger.

Figure 18(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 Fig. 17 does *not* yield this automaton. This automaton can be obtained using more sophisticated variants of constraint automata [29,34], or an encoding technique [52] which can handle context sensitivity.

Figure 18(e) shows the constraint automaton representing the semantics of the `ShiftLossyFIFO1` connector of Fig. 12(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 [5,22–24,28,37,55,56]. Results on equivalence and containment of the languages of constraint automata [25] and failure based equivalences [43] provide opportunities for analysis and optimization of Reo connectors.

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, as explained in the next section.

Variants of the constraint automata model have been devised to capture time-sensitive behavior [11,53,54], probabilistic behavior [20], stochastic behavior [26],

context sensitive behavior [29,34,41], fairness [30,42], resource sensitivity [66], and the QoS aspects [12,13,67,70,71] of Reo connectors and composite systems.

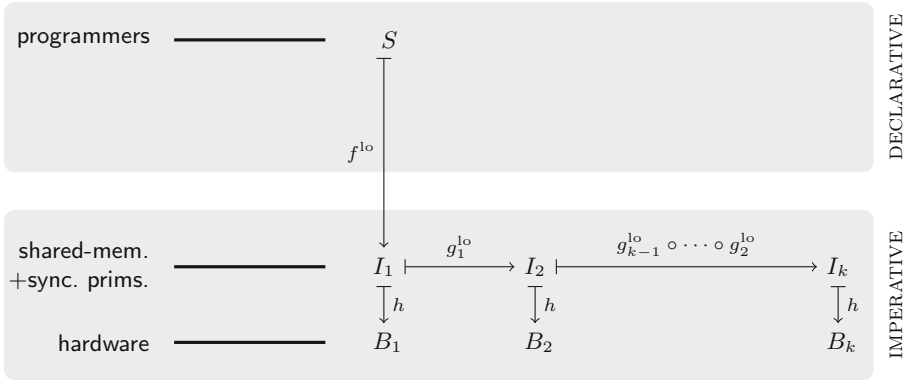**Fig. 19.** From declarative specifications to imperative implementations

6   Compilation

By now, we may have convinced our readers that both (1) exogenous specification of multi-party interaction protocols (regardless of the language in which they are implemented), and (2) high-level languages that support specification of such protocols as composition of primitive interactions (as opposed to in terms of low level communication actions) offer clear software engineering advantages (e.g., programmability, maintainability, reusability, verifiability, etc.). Reo serves as a prime example of a high-level language, based on an exogenous interaction-centric model of concurrency, that demonstrates the viability of raising the level of abstraction in specification of concurrency protocols to where these software engineering advantages can indeed materialize. It seems far less obvious, however, that protocol specifications expressed in such high-level languages can be compiled into efficient and scalable binaries.

In this section, we intend to persuade the reader that in time, sufficiently smart compilers for high-level protocol languages can produce binaries with better performance than binaries produced by compilers for contemporary general-purpose languages that offer the lower-level constructs of traditional models of concurrency. At the core of our argument lies the observation that compilers for such high-level protocol languages can optimize concurrent programs in novel ways inconceivable for compilers that receive lower-level constructs of traditional models of concurrency as their input. In making this argument, first, we need to understand the limitations of compiling protocols coded in lower-level action-based languages as Java and C.

Essentially, to write a concurrent program, concurrent programmers cross a distance between a declarative specification of its protocols and processes

(or threads, components, services, actors, agents, etc.), which abstractly defines *what* must happen, and its imperative implementation, which concretely defines *how* things happen. Today, the processes in such imperative implementations typically interact with each other through actions that manipulate shared-memory protected by classical synchronization primitives, such as locks, semaphores, or monitors.[4] Figure 19 shows our perspective on this approach in terms of three levels of abstraction: (i) the specification interpreted by programmers, denoted by S, (ii) its implementations using shared-memory protected by classical synchronization primitives, denoted by I_i, and (iii) the binaries executed by the hardware, denoted by B_i. In writing their concurrent program, programmers first cross the distance between S and I_1, denoted by arrow f^{lo}. Subsequently, possibly assisted by tools, these programmers may incrementally improve I_1 into implementations $I_2 \ldots I_k$ by applying high-level optimizations to the program logic (e.g., introducing more fine-grained concurrency or replacing data structures with more optimal ones), denoted by arrows $g_1^{\text{lo}} \ldots g_{k-1}^{\text{lo}}$. Finally, a compiler crosses the remaining distance between I_i and B_i, denoted by arrow h.

Figure 19 provides another perspective on the previously identified difficulties with low-level action-based concurrency. Essentially, these difficulties arise from the conceptually long distance between the levels of abstraction of S and I_1, effectively measured by comparing the textual length of specification S with the number of lines of code of its implementation I_1. Intuitively, as this ratio gets smaller, the distance between S and I_1 grows longer, and consequently, the amount of intellectual work that programmers need to perform becomes larger. In practice, it typically requires a substantial effort and significant ingenuity from programmers to define f^{lo} (i.e., to write their concurrent program with action-based concurrency) and to establish $f^{\text{lo}}(S) \sqsubseteq S$ in terms of the low-level code that $f^{\text{lo}}(S)$ consists of (i.e., to establish that f^{lo} faithfully implements S). Hamberg and Vaandrager, for instance, discuss these issues in more detail, from the perspective of teaching concurrency through model checking [39].

Additionally, Fig. 19 also shows that facing a traditional low-level action-based model of concurrency, forces programmers to take responsibility for defining, selecting, and applying every g_i^{lo} (i.e., defining, selecting, and applying optimizations) and, again, for establishing $(g_{k-1}^{\text{lo}} \circ \cdots \circ g_1^{\text{lo}} \circ f^{\text{lo}})(S) \sqsubseteq S$. Ideally, of course, a compiler instead of programmers should perform every g_i^{lo}. But although sixty years of research in compiler technology has resulted in a battery of many important low-level optimization techniques, current compilers typically cannot apply higher-level, "intention-preserving" optimizations to the program logic. For instance, automatic parallelization of general algorithms and data structures remains an open problem to this day [18].

To further illustrate this point, Fig. 20 shows the problem that a low-level compiler faces in applying such high-level "intention-preserving" optimizations. For such a compiler to decide which optimizations it can—and should—apply

---

[4] Of course, in a distributed memory setting, the concurrency primitives are different, but message passing communication primitives used in such settings still constitute an action-based model of concurrency, for which our subsequent argument still holds.
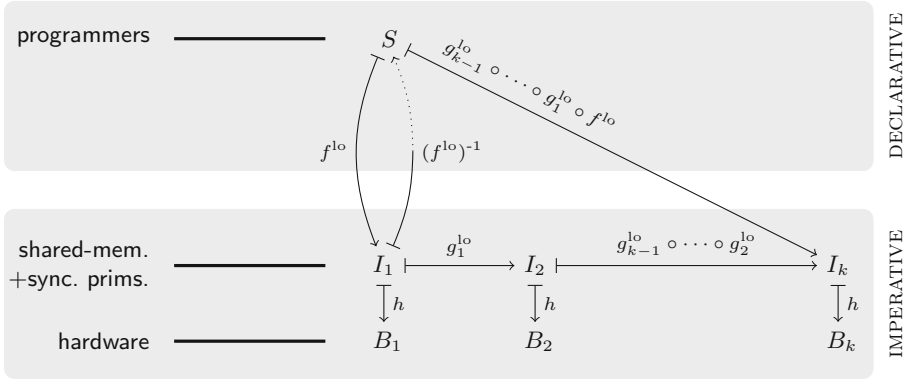
Fig. 20. Irresurrectability of declarative specifications

to which parts of implementation I_1, it essentially needs to reconstruct specification S. Only then, when the compiler knows the *itentions* that programmers had when they wrote I_1, can it decide which portions of the code admit which intention-preserving optimization. In other words, before the compiler can optimize anything, it first needs to apply the *inverse* of f^{lo} to $f^{lo}(S)$ to resurrect the lost *what*, S. Generally, however, the compiler cannot do this: in going from a declarative specification to an imperative implementation, certain information gets irretrievably lost or becomes practically impossible to extract from the resulting code. Consider, for instance, the following C code:

```
int x;
for (int i = 0; i < 10; i++) {
  x = rand();
  a[i] = some_function(x); // without side effects
}
```

If we intended *just* to assign the output of `some_function` to every `a[i]`, for random inputs `x`, a compiler can parallelize the loop. However, if we *additionally* intended the resulting array to have the same content in executions with the same random seed (e.g., to reproduce bugs), a compiler cannot parallelize the loop: in that case, the order of generating random numbers matters. Just from this code, thus, neither a compiler *nor a human* can judiciously decide about loop parallelization; to make that decision, one needs more information.

For more complex programs, as the distance between specifications and their implementations becomes longer, the distance between those implementations and their binaries becomes relatively shorter, leaving less room for a compiler to perform significant high-level "intention preserving" optimizations. Incidentally, the annotations used in some parallelization frameworks (e.g., OpenMP) explicitly preserve information that otherwise gets lost in translation, which the compiler subsequently leverages to produce more optimized binaries. For instance, with OpenMP, a programmer can annotate the loop in the previous C code with the following pragma to inform the compiler that it may parallelize:

```
#pragma omp parallel for private(x)
```

In summary, the distance between high-level declarative specifications of protocols/processes and their low-level imperative implementations using action-based concurrency models hinders concurrent programming in two ways: (1) this distance is too long for average programmers to reasonably write correct code, let alone, correct code that is also efficient, and scalable; (2) the low level of abstraction of the synchronization primitives in which they write their code leaves too small a domain for compilers to perform effective, high-level, intention-preserving optimizations. The latter subsequently forces programmers to take direct responsibility for such optimizations, thereby adding even more complexity to the already daunting task of programmers. To alleviate these issues, programming language designers should provide programmers new, declarative, high-level interaction-based abstractions for implementing parallel programs. In the previous sections, we already argued that languages that offer such constructs, as Reo, can alleviate the first issue. Here, Figs. 19 and 20 give us the right context to argue that such languages also alleviate the second issue.

Figure 21 shows our proposed approach, where $M_1 \ldots M_k$ denote implementations of S in a special, declarative protocol language (imagine Reo). The shorter distance between S and M_1 simplifies programmers' task of writing their parallel program, denoted by arrow f^{hi}, essentially because those programmers need to concern themselves with fewer details (e.g., seemingly nondeterministic scheduling). Moreover, as programmers express their protocols at a high level of abstraction, declaratively, more information about their intentions remains available in the resulting code. A compiler can subsequently leverage this information to generate more optimized binaries. As such, this compiler relieves programmers from the responsibility of manually implementing, and establishing the correctness of, not only low-level optimizations (as current compilers already do) but also high-level intention-preserving optimizations: an application programmer now needs to work out only f^{hi}, after which the compiler takes care of selecting
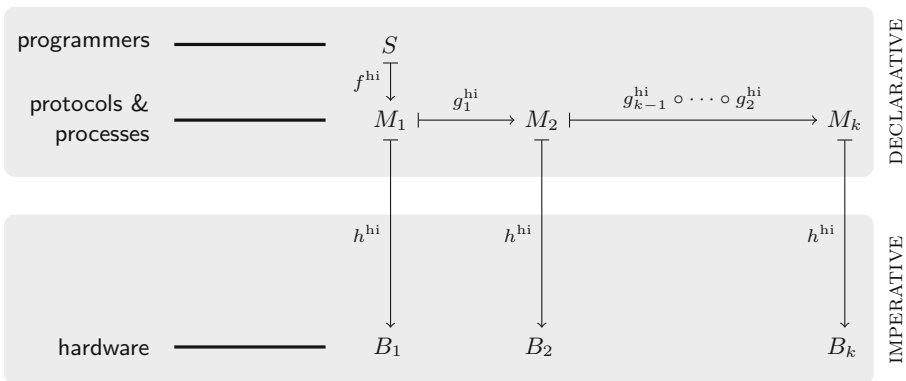

Fig. 21. From declarative specifications to declarative implementations

and applying every applicable g_i^{hi} defined by its designer. This designer, instead of application programmers, should prove the correctness and effectiveness of every g_i^{hi}, and establishing those properties remains a one-shot activity (cf. ad-hoc reasoning about every manually optimized low-level concurrent program). Moreover, because M_i and M_{i+1} reside at a higher level of abstraction than I_i and I_{i+1} do, proving the correctness and effectiveness of g_i^{hi} typically becomes simpler, clearer, and more mathematically elegant than reasoning about the low-level code manipulated by g_i^{lo}. Shortly, we give concrete examples for this claim.

Thus, by offering a new level of interaction-based abstraction to programmers, our proposed approach alleviates the software engineering difficulties of expressing implementations using action-based models of concurrency, by shortening the distance between specifications and their implementations, which in turn makes it more reasonable for programmers to perform the intellectual work required to cross this distance. Perhaps surprisingly, a shorter distance between specification and implementation has another significant advantage: it makes the distance between implementations and binaries long enough for compilers to perform also high-level intention-preserving optimizations, which also ameliorates the difficulties of developing implementations with good scalability and performance. In time, binaries generated by sufficiently smart compilers for high-level protocol languages should outperform binaries of low-level code hand-written by average programmers. In posing this thesis, we feel encouraged by the observation that although the distance between an implementation of a typical sequential program expressed in a conventional imperative languages (e.g., Java or C) and an optimized version of its binary code is also huge, the compiler construction community has still succeeded to develop effective tools for crossing this distance, demanding little or no intellectual effort from programmers. Essentially, we propose to extend that work to high-level protocol languages for concurrent programming. By now, the concrete preliminary results of experiments with our Reo compiler support our thesis and exemplify its feasibility.
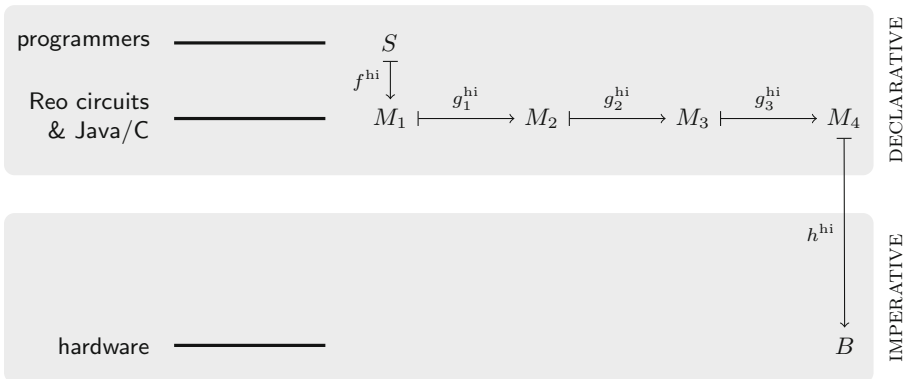

Fig. 22. From declarative specifications to imperative implementations via CAs

Figure 22 shows the instantiation of Fig. 21 in the context of our Reo compiler, which is based on Reo's constraint automaton semantics, presented in Sect. 5. In this instantiation, the programmers' task f^{hi} consists of (i) translating the processes in specification S into Java or C code and (ii) translating the protocol in S into a Reo connector; together, this code and the Reo connector constitute M_1. Our compiler subsequently maps every node and every channel in the Reo connector to its corresponding constraint automaton. This yields a set of "small" automata that collectively represent the connector's semantics. The compiler then translates this set of small automata into Java/C and merges the code so generated with the Java/C code for the processes. An external compiler for Java/C subsequently translates the full code base into a binary.

Our Reo compiler currently applies three high-level optimizations g_i^{hi}.

- g_1^{hi}—*Improving latency* [46]
  The most straight-forward translation of a parallel composition of small automata, which collectively model a connector's semantics, into Java/C works by generating a distinct thread for each of those automata. In this approach, every such thread executes a small state machine for its corresponding automaton, firing transitions as it reaches consensus with the other threads about their collective behavior. The distributed consensus algorithm necessary for achieving such multiparty synchronization, however, costs too much in terms of resources at run-time, which causes transitions to have high firing latency.
  Optimization g_1^{hi} aims at reducing firing latency: instead of translating a parallel composition of small automata to as many threads, g_1^{hi} first computes a single "big" automaton for that composition, similar to parallel expansion in process calculi, and generates only one thread for that automaton. This single thread executes a big state machine, free of other threads to synchronize its behavior with.
- g_2^{hi}—*Improving throughput* [45, 47, 48, 51]
  Although g_1^{hi} reduces firing latency, it does so at the cost of reduced firing throughput: by computing one big automaton out of multiple small automata, g_1^{hi} effectively serializes all parallelism among those small automata. If those small automata have heavy synchronization interdependencies, this is desirable, but if the small automata are more "loosely coupled", such sequentialization may unnecessarily reduce throughput. In that case, at run-time, independent transitions cannot fire in parallel but are artificially serialized.
  Optimization g_2^{hi} aims at improving firing throughput: instead of computing one big automaton for a parallel composition of small automata, it carefully *partitions* that set of small automata into a number of disjoint subsets. Then, for every resulting subset, it composes that subset's elements into a "medium" automaton and generates a thread for that automaton. Every such thread executes a medium state machine, but because of how g_2^{hi} partitions the set of small automata, the consensus algorithm necessary for achieving multiparty synchronization among those threads costs only little in terms of resources at run-time. Consequently, g_2^{hi} balances low latency (i.e., sequentiality) with high throughput (i.e., parallelism).

– g_3^{hi}—*Improving scalability* [49]

Before a thread can fire a transition, it must check the synchronization constraint and the data constraint of that transition. To check the synchronization constraint, a thread inspects all relevant interface nodes for a pending I/O-operation; if at least one of those nodes has no such operation, the transition cannot fire. To check the data constraint, a thread calls a constraint solver to find a solution for that constraint; if no solution exists, the transition cannot fire. Whenever a transition does fire, its executing thread effectively effectuates an interaction among processes. Typically, as the number of processes increases, the number of transitions per medium automaton also increases. Because the thread for such a medium automaton needs to check all its transitions for enabledness (in the worst case), firing a transition requires increasingly more resources as the number of processes increases. This suggests poor scalability.

Optimization g_3^{hi} aims at improving scalability: instead of directly generating a thread for a medium CA, it first *merges* certain distinguished transitions of that automaton into a single transition in a semantics-preserving way. Subsequently, it translates the resulting automaton, with merged transitions, into a thread. This thread executes in the same way as before, but it can check merged transitions for enabledness with a single operation, instead of with one operation per transition. Crucially, to facilitate such combined checks, g_3^{hi} injects optimized data structures for pending I/O-operations on nodes in the generated code. Because not all transitions can be merged in a semantics-preserving way, g_3^{hi} performs static analysis on the transitions of an automaton to determine the extent to which it can introduce such optimized data structures.

We have proved the correctness of the above high-level optimizations in terms of constraint automata (see their respective references, above).

The Java bytecode obtained using our compiler (and an external Java compiler afterward) runs on a JVM as any other Java program. With C, as an extra low-level optimization, we use a framework that allows instructions to be scheduled directly to cores instead of indirectly via the operating system's scheduler [49].

Compilers for low-level languages seem incapable of performing similar optimizations as those in Fig. 22 (i.e., automatic sequentialization, automatic parallelization, and automatic optimization of data structures). As practical evidence, if those compilers would be capable of this, we would have relied on those capabilities of theirs instead of developing optimizations ourselves. More philosophically, we believe that low-level compilers will never be capable of optimizing in this way, simply because they do not have enough information about programmers' intentions (see, e.g., the example of assigning random numbers to array elements, mentioned earlier in this section). Constraint automata, in contrast, retain enough such information to allow more effective high-level optimizations. At the same time, it may be difficult for average programmers to detect when and how optimizations similar to the ones in Fig. 22 may and should be applied manually; compilers for high-level protocol languages alleviate this burden.
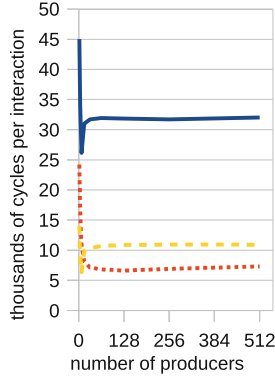
Fig. 23. Earlier performance results [50] (Color figure online)

For some protocols, the high-level intention-preserving optimizations in Fig. 22 already allow our compiler to generate code that can compete with code written by a competent programmer [50]. Figure 23 shows one of our most promising achievements so far. It shows the performance of three implementations of a k-producers-single-consumers protocol, for $k \in \left\{2^i \mid 2 \leq i \leq 9\right\}$: one naive hand-written implementation in C (blue, solid line), one hand-crafted optimized implementation in C (yellow, dashed line), and one implementation expressed in Reo and compiled via cas into C (red, dotted line). In every round of this protocol, every producer sends one datum to the consumer. Once the consumer has received a datum from every producer, in any order, it sends an acknowledgment to the producers, thereby signaling that the consumer is ready for the next round. To measure just the performance of the protocol, we did not give the producers and the consumers real computational tasks (i.e., the producers sent only dummy data). This example shows that already our current compilation technology is capable of generating code that can compete with—and in this case even outperform—carefully hand-crafted code. Surely, our technology is not yet mature enough to always achieve such positive results. Nevertheless, this example offers preliminary evidence that programming protocols among threads using high-level, interaction-based constructs and abstractions can result in equally good—or better—performance as compared to hand-crafted code using conventional low-level, action-based models of concurrency.

The obvious superficial "performance comparison" depicted in Fig. 23 may say as much about the effectiveness of our optimization techniques, as it does about the competency of the C programmer who produced the hand-crafted version of the protocol code of this application. However, below this surface, lies a more crucial fundamental point that is independent of the competency of any individual programmer, or the precise factor by which our optimization techniques potentially can or currently do outperform hand-crafted code that a programmer can (even hypothetically) produce. Crucial to this benchmark is the fact that the task assigned to the programmer restricted him to use concurrency

constructs available in contemporary programming languages, such as Java or C (in this case p-threads). On the other hand, our Reo compiler bypasses this level of abstraction (and the coarser-grained, OS-level scheduling inefficiencies that it entails) and generates code using finer-grained constructs *below* the OS-level and the concurrency constructs that it supports. From this perspective, comparing the performance of the two versions of the code is even unfair, because the statement of his task assignment prevents the programmer from using lower-level constructs to directly hand-craft code similar to (or even better than) what our Reo compiler produces. But precisely this *unfairness* constitutes the crux of our argument in this section.

There are two conceivable ways to make such a comparison fair, i.e., produce code using constructs that are "fairly comparable" to the constructs that our Reo compiler uses to produces its code: (1) allow the programmer to directly code below the level of p-threads and OS; or (2) develop tools that take p-threads level code written by a programmer and produce more optimized code.

Option 1, i.e., removing the artificial barrier of programming at the level of p-threads, is certainly possible. However, programming below p-threads and OS-level sharply raises the level of expertise required by a programmer to code directly at such a low level, and dramatically increases the size and the complexity of the resulting code. Higher competency requirements and increased size and complexity of code, in turn, sharply reduce the number of competent individuals who qualify to perform such programming assignments, and dramatically lower the likelihood of success of those who undertake such daunting tasks. Besides, applications that directly use constructs below p-threads or OS abstractions become highly brittle and non-portable, as they rely on constructs that most likely do not exist verbatim on other platforms, or even on a future upgrade of their original platforms.

Option 2 requires developing tools that can reconstruct the intentions behind the p-threads constructs used to encode a protocol (fragment). As a concrete example, a single transition in a constraint automaton may declare a complex multi-party synchronization. By the time that a programmer expresses this intention in terms of semaphores, locks, guards, communication primitives, and data structures, and intersperses its resulting code with other fragments of code that are not directly related to this specific (multi-party synchronization) intention, it becomes extremely difficult, if not theoretically impossible, for any tool to reconstruct the original intention. Not having this information prevents a tool from performing intention-preserving optimizations to generate lower-level code that can more efficiently implement an application-specific multi-party synchronization.

Offering programmers higher-level protocol specification languages, such as Reo, which directly capture and retain more of the intentions behind protocol fragments, seems like a very promising alternative. Our work on Reo and our preliminary experiments with our Reo compiler suggest this approach is a viable alternative.

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 connectors to illustrate the flavor of 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 scaled or modified, even at run time.

In addition to software engineering advantages, high-level languages to specify multi-party exogenous interaction protocols, such as Reo, have advantages with respect to performance as well: as evidenced by our Reo compiler, compilers for such high-level languages can perform optimizations that compilers for lower-level languages cannot apply.

References

1. CADP home page. http://www.inrialpes.fr/vasy/cadp/
2. Extensible 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. Reo home page. http://reo.project.cwi.nl
5. Vereofy home page. http://www.vereofy.de/

6. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
7. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
8. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Sci. Comput. Program. **55**(1–3), 3–52 (2005)
9. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011)
10. Arbab, F., Aştefănoaei, L., de Boer, F.S., Dastani, 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)
11. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. Softw. Syst. Model. **6**(1), 59–82 (2007)
12. 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)
13. Arbab, F., Chothia, T., van der Mei, R., Meng, S., Moon, Y.-J., Verhoef, C.: From coordination to stochastic models of QoS. In: Field and Vasconcelos [35], pp. 268–287
14. Arbab, F., Kokash, N., Meng, S.: Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 108–123. Springer, Heidelberg (2008)
15. 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)
16. Arbab, F., Meng, S.: Synthesis of connectors from scenario-based interaction specifications. In: Chaudron, M.R.V., Ren, X.-M., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 114–129. Springer, Heidelberg (2008)
17. 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)
18. Arvind, D.A., Pingali, K., Chiou, D., Sendag, R., Yi, J.: Programming multicores: do applications programmers need to write explicitly parallel programs? IEEE Micro **30**(3), 19–33 (2010)
19. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, Cambridge (1990)
20. Baier, C.: Probabilistic models for Reo connector circuits. J. Univers. Comput. Sci. **11**(10), 1718–1748 (2005)
21. 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)
22. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field and Vasconcelos [35], pp. 247–267
23. 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, Part II. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010)

24. 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)
25. 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)
26. 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)
27. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Inf. Control **60**, 109–137 (1984)
28. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. Electr. Notes Theor. Comput. Sci **215**, 209–226 (2008)
29. Bonsangue, M.M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Field and Vasconcelos [35], pp. 184–203
30. 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)
31. 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)
32. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Sci. Comput. Program. **66**(3), 205–225 (2007)
33. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Sci. Comput. Program. **76**(8), 681–710 (2011)
34. Costa, D.: Formal models for context dependent connectors for distributed software components and services. Ph.D. thesis, Vrije Universiteit Amsterdam (2010). http://dare.ubvu.vu.nl//handle/1871/16380
35. Field, J., Vasconcelos, V.T. (eds.): COORDINATION 2009. LNCS, vol. 5521. Springer, Heidelberg (2009)
36. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer, Berlin (1999)
37. 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)
38. 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)
39. Hamberg, R., Vaandrager, F.: Using model checkers in an introductory course on operating systems. Oper. Syst. Rev. **42**(6), 101–111 (2008)
40. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
41. 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)
42. Izadi, M., Bonsangue, M.M., Clarke, D.: Büchi automata for modeling component connectors. Softw. Syst. Model. **10**(2), 183–200 (2011)

43. Izadi, M., Movaghar, A.: Failure-based equivalence of constraint automata. Int. J. Comput. Math. **87**(11), 2426–2443 (2010)
44. Jongmans, S.-S., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
45. Jongmans, S.-S.T.Q., Arbab, F.: Global consensus through local synchronization. In: Canal, C., Villari, M. (eds.) ESOCC 2013. CCIS, vol. 393, pp. 174–188. Springer, Heidelberg (2013)
46. Jongmans, S.-S., Arbab, F.: Modularizing and specifying protocols among threads.In: Proceedings of PLACES 2012. EPTCS, vol. 109, pp. 34–45. CoRR (2013)
47. Jongmans, S.-S., Arbab, F.: Toward sequentializing overparallelized protocol code. In: Proceedings of ICE 2014. EPTCS, vol. 166, pp. 38–44. CoRR (2014)
48. Jongmans, S.-S., Arbab, F.: Can high throughput atone for high latency in compiler-generated protocol code? In: Proceedings of FSEN 2015. Springer (in press)
49. Jongmans, S.-S.T.Q., Halle, S., Arbab, F.: Automata-based optimization of interaction protocols for scalable multicore platforms. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 65–82. Springer, Heidelberg (2014)
50. Jongmans, S.-S., Halle, S., Arbab, F.: Reo: a dataflow inspired language for multicore.In: Proceedings of DFM 2013, pp. 42–50. IEEE (2014)
51. Jongmans, S.-S., Santini, F., Arbab, F.: Partially-distributed coordination with Reo.In: Proceedings of PDP 2014, pp. 697–706. IEEE (2014)
52. Jongmans, S.-S.T.Q., Krause, C., Arbab, F.: Encoding context-sensitivity in reo into non-context-sensitive semantic models. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 31–48. Springer, Heidelberg (2011)
53. Kemper, S.: SAT-based verification for timed component connectors. Electr. Notes Theor. Comput. Sci. **255**, 103–118 (2009)
54. 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)
55. 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)
56. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Electr. Notes Theor. Comput. Sci **175**(2), 19–37 (2007)
57. Koehler, C., Lazovik, A., Arbab, F.: ReoService: coordination modeling tool. In: Krämer et al. [64], pp. 625–626
58. Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: SAC 2010: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2406–2413. ACM, New York (2010)
59. 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)
60. 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)
61. 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)

62. 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)
63. 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)
64. Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.): ICSOC 2007. LNCS, vol. 4749. Springer, Heidelberg (2007)
65. Lazovik, A., Arbab, F.: Using Reo for service coordination.In: Krämer et al. [64], pp. 398–403
66. 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)
67. Meng, S., Arbab, F.: QoS-driven service selection and composition. In: Billington, J., Duan, Z., Koutny, M. (eds.) ACSD, pp. 160–169. IEEE (2008)
68. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
69. Milner, R.: Elements of interaction - turing award lecture. Commun. ACM **36**(1), 78–89 (1993)
70. Moon, Y.-J.: Stochastic models for quality of service of component connectors. Ph.D. thesis, Leiden University (2011)
71. Moon, Y.-J., Silva, A., Krause, C., Arbab, F.: A compositional semantics for stochastic Reo connectors. In: Mousavi, M.R., Salaün, G. (eds.) FOCLASA. EPTCS, vol. 30, pp. 93–107 (2010)
72. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York (2001)
73. 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)
74. Wegner, P.: Coordination as comstrainted interaction (extended abstract). In: Hankin, C., Ciancarini, P. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 28–33. Springer, Heidelberg (1996)