Proper Protocol

Farhad Arbab[1,2(✉)]

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

Abstract. Treating interaction as an explicit first-class concept, complete with its own composition operators, leads to a model of concurrency that allows direct specification and manipulation of protocols as proper mathematical objects. Reo [2,5,6,8] serves as a premier example of such an interaction-centric model of concurrency.

In this paper, we peruse Reo and explain how its model of protocols as encapsulated, reusable constructs facilitates their fulfilling of the more prominent role slated for them in engineering of modular, verifiable, scalable concurrent software. We also explore clues enlaced with some recent results of our ongoing work on compiling Reo protocol specifications into efficient executable code, which sketch a promising perspective for future work on high-level protocol specification languages.

1 Preamble

For the bulk of the time that Frank de Boer and I have been colleagues at CWI and Leiden, my work has focused on concurrency, coordination, and Reo, and Frank has been working on concurrency, object orientation, formal verification, and many other topics. Nevertheless, Frank's impact on Reo goes beyond his direct contributions through formal collaborations on projects and his coauthorship of papers. Through discussions and by his interest and his questions, Frank has helped me—as well as many of our colleagues—to focus and refine our understanding, and even chart our course into new projects.

2 Introduction

Today's low-cost multicore commodity hardware has made scalable parallel computing platforms affordable. Offering many processor cores on the same chip, cheap threading with fast communication and shared memory, these platforms can potentially accommodate applications that requires massively concurrent computing. Nevertheless, full utilization of the enormous potential offered by such platforms in real-life applications seems to lag dramatically behind. The striking gap between the potential massive concurrency offered by these platforms and their practical uptake raises a perhaps heretical question: do we even

E. Ábrahám et al. (Eds.): de Boer Festschrift, LNCS 9660, pp. 65–87, 2016.
DOI: 10.1007/978-3-319-30734-3_7

need such massively concurrent platforms? More specifically, what types of applications can actually benefit from such massively concurrent platforms, and by how much?

A most emphatically positive answer to the first question may provide an answer to the second by identifying an auspiciously significant class of important applications that can benefit by a substantial factor. However, such a propitious outcome of this inquiry, in turn, raises another question: if massively concurrent systems have important practical applications and computing platforms do exist to provide them, then what has hindered extensive uptake of these platforms to accommodate those applications?

We argue that an emphatically positive answer to the first question is indeed justified. A vast number of important problems can indeed use large-scale coarse grain concurrency, at least in principle. However, conspicuously missing are effective techniques for developing scalable concurrent software that turns the raw computing power of massively concurrent multicore platforms into effective applications that solve those problems.

The growing importance of applications that involve huge volumes of data and peta-scale graphs of their inter-relations, make the need for programming techniques to harness the massive concurrency offered by multicore platforms ever more vivid. To find what has hindered extensive development of massively concurrent applications we must look into the inadequacies of contemporary programming constructs and models for concurrency. These inadequacies stem from the fact that, ironically, *concurrency protocols* have not received the proper attention that they deserve in the classical work on concurrency.

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 [30], CCS [49], the π-calculus [50,53], further process algebras [11,16,23], and the actor model [3] 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-centric* models of concurrency. All these models provide constructs for the direct specification of *things that interact*, rather than a direct specification of *interaction* (protocols). Consequently, in these formalisms (a protocol that specifies an intended) interaction 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.

Our work on Reo shows that one can formally treat interaction as an explicit first-class concept, complete with its own composition operators. Several significant advantages ensue from such an *interaction-centric* model of concurrency. Treating protocols as proper mathematical objects expressed as encapsulated syntactic constructs, explicitly separates them from computation code of applications, which simplifies software development by adhering to the principle of separation of concerns. Separation of protocols from computation allows

formal verification and analysis of protocols in isolation from any application code. As concrete encapsulated formal constructs, one can reuse such formally verified protocols, verbatim—perhaps out of a library—in different applications. Moreover, one can directly compose simpler (verified) protocols into arbitrarily more complex protocols, which allows compositional verification of the resulting more complex protocols. Finally, although it may superficially seem counter-intuitive, an interaction-centric model of concurrency, such as Reo, opens up a vast field of opportunities to refine information-rich, high-level models of protocols into efficient executable code whose performance can compete with and even beat that of carefully hand-crafted code.

3 The Bounty of Concurrency

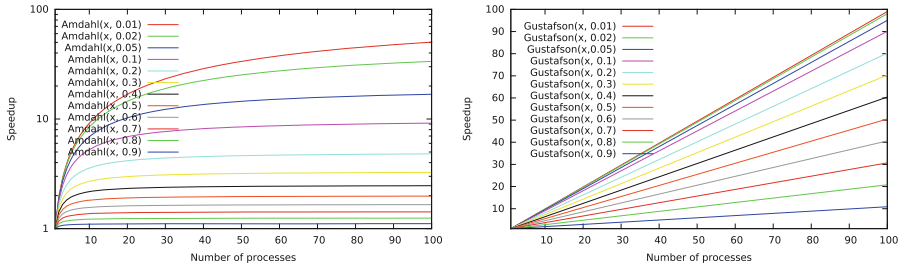
The extent to which a solution to a problem can benefit from concurrency depends on the amount of concurrency inherent in that problem. The famous computer architect Gene Amdahl[1], had quantified this message in what has become known as Amdahl's law [4].

Amdahl's Law. An application consists of an inherently sequential part and a potentially concurrent part. Let a designate the time that it takes to execute the sequential part on a single processor, and b the time that it takes to execute the potentially concurrent part on a single processor. Thus the total execution time of this application on a single processor is $T(1) = a + b$. Generously ignoring all overhead, throwing n processors at this application can speed up only its potentially concurrent part by a factor of n. Thus, the total execution time of this application on an n-processor machine is $T(n) = a + b/n$. Therefore, the speedup that we can expect from running this application on an n-processor machine compared to running it on a single processor is bound by $S_{Amdahl}(n) = \frac{T(1)}{T(n)}$. Define $\alpha = a/(a+b)$ and we obtain Amdahl's law expressing the upper-bound for the speedup of an application running on n processors compared to its execution time on a single processor, in terms of its inherently sequential fraction, α:

$$S_{Amdahl}(n, \alpha) = \frac{1}{\alpha + \frac{1-\alpha}{n}} \tag{1}$$

Figure 1(a) shows the graph of speedup (on logarithmic scale) according to Amdahl's law as a function of number of processors, for a range of α values from 0.01 to 0.9. This graph puts a discouraging damper on the enthusiasm about the speedup of applications on parallel machines. If 50 % of an application is inherently sequential, its execution on a 2-processor machine speeds up by a factor of 1.33 and this "linear" speedup tapers off quickly to 1.66 on a 5-processor machine, improving to nearly 2 only on a 100-processor machine. If only 10 % of an application is inherently sequential, its execution improves almost linearly by

[1] While this paper was under review, Gene Amdahl [16 November 1922 – 10 November 2015] passed away.



(a) Amdahl's law (b) Gustafson's law

Fig. 1. Amdahl's and Gustafson's laws

adding up to 5 processors, but this improvement tapers off to a speedup of only 5.26 with 10 processors, 6.89 with 20, and 9.17 with 100 processors. Even with an infinite number of processors, this application speeds up by only a depressing factor of 10! For an application 99 % of which is inherently concurrent, nearly-linear speed up lasts only up to about a dozen processors; 20 processors yield a 16.80 speedup, which tapers off to only 50.25 with 100 processors.

Amdahl's law is in fact not as depressing as it may seem, because it simply states an obvious fact: that there is just so much juice that you can extract out of an orange, no matter how long and hard you press it (even if we ignore the overhead of the juice that gets trapped and goes to waste in the pulp). An application that spends only 10 % of its time executing its inherently sequential part has no more than a ten-fold juice of speedup to extract, even if you press it by the computational force of an infinite number of processors. In practice, you may be happy with a 9-fold speedup of this application on a 100-processor machine, or settle for a 5-fold speedup with only 10 processors, and let the remaining speedup juice go to waste with the pulp, because obtaining this remaining speedup is simply not worth the cost of its extraction.

Is this the best we can hope to reap from the bounty bestowed by massively concurrent hardware?

Gustafson's Law. Amdahl's law gives a bound for how much juice we can extract from a specific individual orange, i.e., how much faster we can run an application that solves a *fixed-size* problem on a multiprocessor machine. Amdahl's law, however, does not limit our ability to quench our thirst for more orange juice: we can simply juice bigger (amounts of) oranges!

A very important class of applications in concurrency involves solving problems whose sizes can increase arbitrarily. What matters in these applications is not so much speeding up the solution of a specific instance of such a problem (e.g., mining graphs of a given size) on a multiprocessor machine. We may already be content (if not happy) with the execution speed of this solution on a k-processor machine. The purpose of employing more than k processors in such applications is to solve larger-sized instances of the same problem (e.g., mining

proportionally larger graphs) in still reasonable time. Bigger-size problems, thus, provide arbitrarily bigger (amounts of) oranges to juice!

Gustafson revisited Amdahl's law to accommodate precisely this class of applications [26], which we call *scalable*. Let $T_k(n)$ denote the execution time of a scalable problem of size n on a k processor machine. A scalable application also contains an inherently sequential part, whose execution on a single processor takes a time units. The potentially concurrent part of such an application has a repetitive structure that scales directly with the size of the problem. Let b be the sequential execution time of the potentially concurrent part of this application, solving the size-1 instance of the problem. The total execution time of the application for the size-1 instance of the problem on a single processor, then, is $T_1(1) = a + b$, and the execution time of a size-n instance of the problem on a single processor machine is $T_1(n) = a + n \times b$.

With more processors, we can parallelize the potentially concurrent part of solving a larger instance of the problem. Thus, ignoring all overhead, the execution time of a size-n instance of the problem on an n-processor machine is $T_n(n) = a + b$, which means $T_n(n) = T_1(1)$. Defining $\alpha = a/(a + b)$, as before, we get Gustafson's law for speedup:

$$S_{Gustafson}(n, \alpha) = n - \alpha \times (n - 1) \tag{2}$$

Figure 1(b) shows the graph of speedup for scalable problems according to Gustafson's law as a function of number of processors, for a range of α values from 0.01 to 0.9. It seems that at least for scalable problems, Gustafson's law rescues usefulness of concurrency from the grim grip of Amdahl's law.

Superficially, the two graphs in Fig. 1 seem to contradict each other: for every value of α, Fig. 1(a) establishes a strict asymptotic limit less than n for speedup, whereas Fig. 1(b) shows that speedup increases linearly in n, without bounds, at an α-dependent slope. In fact, far from contradicting Amdahl's law, Gustafson's law complements it. For scalable applications, as we increase n, we change the application by increasing the size of the problem that it solves and thereby increase the amount of concurrent juice that it contains. As a result, the ratio of the inherently sequential part of the application to its total execution time on a single processor shrinks, and the application moves up the rungs of the ladder of α curves in Fig. 1(a).

Let $\delta(n) = a/(a + n \times b)$ designate the fraction of the inherently sequential part of a scalable application of size n. Rewriting $\delta(n)$ in terms of α, we get:

$$\delta(n) = \frac{\alpha}{\alpha + (1 - \alpha) \times n} \tag{3}$$

Equation 3 shows that for a fixed α, as n grows, $\delta(n)$ diminishes, endowing more concurrency juice to the application, which moves the application up the ladder of the curves in the graph of Amdah's law in Fig. 1(a). In fact, substituting $\delta(n)$ for α in Eq. 1, Amdahl's law yields $S_{Amdahl}(n, \delta(n)) = 1$, which shows that for scalable problems, as we increase the number of processors from n to $n + k$ to match the increase of the problem size from n to $n + k$, the quantity $\delta(n)$

diminishes exactly such that we obtain no "real speedup" gain by Amdahl's law: *all* extra concurrency provided by the additional k processors goes to solving the k-size larger problem. This observation suggests that perhaps more usefully, we can think of Gustafson's law not so much as a measure of speedup, but rather a measure of *scalability* of a scalable application.

Communication Overhead. Both Amdahl's and Gustafson's laws mean to express upper bounds, and thus they ignore all overhead. Obviously, the interaction protocol that enables communications among concurrent chunks of the application greatly influences this overhead. To account for protocol overhead, we revise Gustafson's equation for the execution time of a size-n scalable problem on an n-processor machine as $T_n(n) = a + b + c$ where c is the extra time that the application takes to complete because of protocol overhead.

For $n > 1$, every parallel computation fragment executes a number of communication operations. The extra delays required to complete these operations collectively comprise $c = f(n)$, which means speedup of a scalable application is:

$$S(n, \alpha) = \frac{n - \alpha \times (n - 1)}{1 + \frac{f(n)}{T_1(1)}} \tag{4}$$

For any application, α and $T_1(1)$ are constants. The nominator in Eq. 4 is linear in n. The effectiveness of the speedup (or, scalability) of an application, then crucially depends on the nature of its protocol overhead function $f(n)$. A good linear protocol yields only a constant speedup (i.e., no scalability), and even a quadratic $f(n)$ quickly dampens scalability by $1/n$ as n increases.

So, where exactly in a concurrent application can we find its communication protocol that has such a significant impact on its performance and scalability?

4 Where's Waldo?

In a modern well-structured program, we can easily locate a segment of code that implements some computation function, e.g., *FourierTransform*, or a computation construct such as the abstract data type *stack*. These implementations, of course, use concrete algorithms and data structures. For instance, the implementation of *stack* may use a linked list data structure. Because they are so easy to locate, if desired, we can readily replace the implementation code for *FourierTransform* or *stack* with the piece of code for some alternative implementation of these computation constructs. For instance, we can easily replace the linked-list implementation of *stack* with an array implementation of *stack* to improve the performance of an application. If the application software is indeed well-structured (e.g., *stack* is implemented as a class in an object-oriented language), this implementation code swapping will be completely invisible to the rest of the software, regardless of how often or intensively it uses various incarnations of *stack*. A more efficient implementation of an abstract data type or a computation function simply improves the overall performance of

the application, without requiring any modification to the rest of the software. Moreover, to scale up a well-structured program using a stack of size k to one that needs a stack of size $2k$, all we need to do is change the value assigned to some identifier from k to $2k$.

Programming language constructs and abstractions, along with techniques for their efficient compilation, have dramatically advanced in the last half-century, to the extent that we can now program at the level of (parametric) types, classes, objects, mathematical functions, monads, or Horn clauses, when appropriate, and obtain executable code whose performance competes with—indeed often beats—that of code written by even better-than-average programmers in some low-level language. It is precisely these advances that, among other things, make it easy to carry out the above mentioned software modifications so painlessly.

Protocols constitute no less significant a concept in concurrent applications than functions, types, and other computational constructs, and variants of concrete implementations of protocols have an least equally significant impact on the performance of a concurrent application. Moreover, as we saw in Sect. 3, protocol (overhead) plays a crucial role in the scale-up of scalable problems. Given the significance of protocols and the long history of concurrency, one would expect—rather naively—to find in modern software high-level *protocol constructs* (as counterparts to constructs for types, classes, etc.), that make, e.g., scaling up a two-producer-one-consumer protocol to a k-producer-one-consumer protocol as easy as changing the value assigned to some identifier in its implementation code from 2 to k. Or—even more naively—that to change one implementation of a two-producer-one-consumer protocol with another, all that should be necessary is to swap the two pieces of code for their respective implementations, without any change to the rest of the software. Perplexingly, neither of these software modifications is so painless today!

Programming constructs and models for concurrency have essentially stagnated in the past half-century. Algorithmic skeletons [25] represent an attempt to facilitate development of better structured concurrent programs by offering encapsulated protocol skeletons that programmers can flesh out to suit the specifics of their applications. Several skeleton libraries exist. However, although useful in practice when a problem readily fits the design patterns of available skeletons, algorithmic skeletons have not given rise to a formal model of encapsulated, composable protocols analogous to types, objects, and classes. Transactional memory [29,47,54] represents another attempt to simplify concurrent programming by providing *transaction* as a syntactic construct for high-level mutual exclusion. Although a transaction can qualify as a protocol, transactions often necessarily contain application specific computation code, which makes them *impure* (non-reusable). Moreover, treating every protocol as a transaction can lead to over-sequentialization, and this model does not provide adequate means to derive more complex (than single transaction) protocols through structural composition of other protocols (transactions). In contrast to advances in abstractions and constructs for sequential programming, no real abstract *protocol constructs* have evolved. Processes, threads, locks, semaphores, contrivances

for mutual exclusion, monitors, rendezvous, etc., of roughly 50 years ago comprise all programming constructs we have to express protocols in our modern software.

The pervasive integration of computing and interaction in so many aspects of our lives today has vastly expanded the number of applications that require scalable complex protocols. Meanwhile, advances in processor, memory, and communication hardware have made leaps and bounds in the past 50 years to provide suitable hardware to accommodate these applications. Software technology must develop code that transforms the raw power of available hardware into concurrent applications that embody those required scalable complex protocols. Stagnation of programming constructs and models for concurrency has created a stifling bottleneck in development of these applications. Between the two expanding domains of *necessity* and *possibility*, software engineers are left stranded to fend for themselves, armed with nothing more than the same 50-year-old cumbersome concurrency constructs, and their own wits. In this sense, our arsenal of 50-year-old concurrency programming constructs is dramatically less adequate for our software engineering needs of today than it was 50 years ago.

Finding what constituted a *stack* (just as an example) in a typical "well-structured" Fortran IV or PL/I code of early 1970's required as much time and mental effort as finding Waldo[2]—and it was far less entertaining. The mere act of locating what constitutes a protocol in a typical well-structured concurrent application of today often requires substantially more time, effort, and expertise than was required to find a *stack* in a Frotran IV or PL/I application of the 1970's; and replacing the implementation of this protocol or scaling it up often cascades numerous prohibitively intrusive intricate changes throughout the entire software.

5 Interaction-Centric Concurrency

Traditional action-centric models for concurrent programming embed within the sequential programming paradigm a befitting selection of primitives such as locks, semaphores, monitors, send/receive, message passing, rendezvous, etc., for programmers to manifest an interaction protocol contingent on the control flows of disparate sequential threads, that run under an implied nondeterminism on the order of their execution. This dispersion of interaction-inducing actions makes protocols nebulous, intangible, and ephemeral, which explains why even identifying the constructs that constitute a protocol in an application programmed in such models often becomes a non-trivial challenge.

The dataflow paradigm provides an alternative perspective on concurrent programming. It liberates the manifestation of interaction protocols from the control flows of sequential threads, expressing them instead as concrete graphs that make the nondeterminism of their execution explicitly evident. The classical works on dataflow programming pioneered by Kahn [45,46], Dennis [22],

[2] In Martin Handford's 1980's popular books of double-page illustrations, that challenged readers to locate a certain Waldo character "hidden" in plain sight in a crowd.

and Arvind [9] serve as inspiring early examples of interaction-centric models of concurrency: abstracting away the semantic content of computation nodes in such a dataflow graph leaves a structure behind that explicitly represents a concrete interaction protocol. One can easily compose protocols by splicing their graphs together. Because the edges in these specific dataflow graphs represent FIFO communication links, these protocols cannot directly express synchrony. The need for synchrony in concurrency, especially in real-time and embedded systems, led to the development of synchronous languages [15,17,21,24], where edges represent synchronous communication. Ptolemy [19,48] allows hierarchical composition of graphs each representing a synchronous or asynchronous interaction among actors, to model heterogeneous systems.

In the world of sequential programs, with the formal semantics of a function as a black-box that transforms its input to its output, the semantic equivalence of two functions is a congruence, i.e., given two equivalent functions, one can always replace the other. In the world of concurrent programs, such semantic equivalence is not a congruence, i.e., given two concurrent computation units whose functions are equivalent, one cannot always replace the other [18]. This observation, known as the Brock-Ackerman anomaly, shows that interaction requires a more expressive formal semantics enriched by a notion of time, to discern differences between otherwise equivalent units of computation that arise out of alternative orders of their execution. Some dataflow models suffer from this anomaly, and some avoid it by imposing restrictions.

Such earlier work as above has inspired our notion of interaction-centric concurrency, and our work on Reo builds upon and extends this work. More recent work on BIP [10,14], multiparty session types [20,32], Scribble [31,55], and Pabble [51,52] represent other examples of interaction-centric models that to various degrees of expressiveness and generality make protocols concrete central objects of discourse.

Treating interaction as a full-fledged first-class concept requires a model that offers (1) an explicit, direct, concrete representation of 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. A most primitive interaction specifies a *relation* between two communication actions, e.g., a send and a receive. For instance, such a relation may state that the two actions must happen synchronously, or that one (e.g., the read) must necessarily happen strictly some time after the other has completed. This specification is oblivious to the actor entities that perform such communication operations; all that matters is that the specified relation holds on the timing and the contents of the data exchanged by those operations. Such specifications quite naturally accede a formal representation as *constraints*, which come equipped with *relational composition* that allows constructing more complex constraints out of simpler ones.

Protocols as Connectors. Concretely, we regard a protocol simply as a constraint, which declaratively specifies *what* must hold in terms of a relation, disregarding *how* it can hold. Expressed as constraints, pure protocols become first-class,

tangible, reusable encapsulated constructs in their own right. As concrete software constructs, such protocols can manifest as architecturally meaningful *connectors* that portrayed graphically, resemble a generalization of dataflow graphs where nodes have fixed semantics but each edge represents an arbitrary interaction relation.

Components. In an interaction-centric model of concurrency, a computational process (or thread), or *component* is written in any conventional programming language, such as C, C++, Java, etc. The only means of communication of a component with its outside world is through *blocking I/O operations* that it may perform exclusively on its own *ports*. Inter-component communication is possible only by mediation of connectors, which implement interaction protocols, outside of the components.

If `i` is an input port of a component, C, there are only two operations that C 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`; or (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 component, there are only two operations that the component 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`; or (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`.

6 Overview of Reo

We have used the interaction-as-constraint perspective described above to formalize an interaction-centric model of concurrency wherein every interaction protocol is a constraint obtained as a (relational) composition of a small set of simple binary constraints. This model serves as the formal foundation of a domain-specific language (DSL), called Reo [2,5–8], for programming concurrency protocols that manifest as connectors. Complex connectors in Reo are constructed as a network of primitive binary connectors, called *channels*.

We summarize only the main concepts in Reo here. Further details about Reo and its semantics can be found in cited references. Tool support for Reo consists of a set of Eclipse plug-ins that together comprise the Extensible Coordination Tools (ECT) visual programming environment [1].

Channels. 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

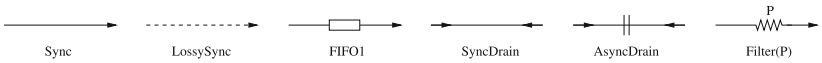


Fig. 2. A typical set of Reo channels

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.

A very small set of channels, each with very simple behavior, suffices to construct useful Reo connectors with significantly complex behavior. Figure 2 shows a common set of primitive channels often used to build Reo connectors. Readers can find intuitive and formal definitions of the behavior of these channels in various Reo literature, e.g. [7].

Nodes. Complex connectors are constructed by composing simpler ones by *joining* channel ends together in *nodes*. A Reo node is a logical place where channel ends coincide and coordinate their dataflows as prescribed by the *type* of the node. Figure 3 shows the three possible node types in Reo. A node is either *source*, *sink*, or *mixed*, depending on whether all its coincident channel ends consist of source ends, sink ends, or a combination of the two. Unlike channels, Reo defines a fixed semantics for (i.e., the constraints on the dataflow through) its nodes.

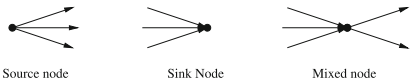


Fig. 3. Reo nodes

The source and sink nodes of a connector are collectively called its *boundary nodes*. Boundary nodes define the interface of a connector. Components attach to the boundary nodes of a connector and interact anonymously via the `get` and `put` operations mentioned in Sect. 5 with each other through this interface.

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 nontrivial. Various models for the formal semantics of Reo have been developed (most, variants that fall within a small number of main families), each to serve some specific purpose, e.g., animation, verification, and code generation; a comprehensive overview of these semantic models appears elsewhere [34].

7 Examples

Consider a simple concurrent application with two producers, which we designate as Green and Red, and one consumer. We want the consumer to repeatedly

obtain and display the data made available by the Green and the Red producers, alternately. In spite of its apparent conciseness, the last sentence does *not* precisely specify a single concrete protocol. In this section, we present a number of protocols to implement different versions of the alternating producers and consumer example. These examples illustrate that using Reo it is trivial to (1) change the protocol of an application, without altering any of its processes, or (2) scale the specification of a protocol to accommodate $k > 2$ producers.

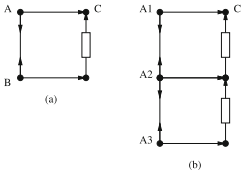


Fig. 4. Alternators

The connector in Fig. 4(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. Subsequent take operations at C obtain the data items written to $A, B, A, B, ...$ The connector in Fig. 4(b) is obtained by replicating the one in Fig. 4(a). It delivers the data items obtained from $A1$, $A2$, and $A3$, through C, in that order.

We can compose a version of our alternating producers and consumer example by attaching the output ports of the Green and Red producers to nodes A and B of the connector in Fig. 4(a), respectively, and the input port of the consumer to its node C. The protocols of the connectors in Fig. 4 synchronize their producers in each round. Whether or not this is a desirable property, of course, depends on the application. Our original specification of this example allows this protocol, as well as many other alternatives.

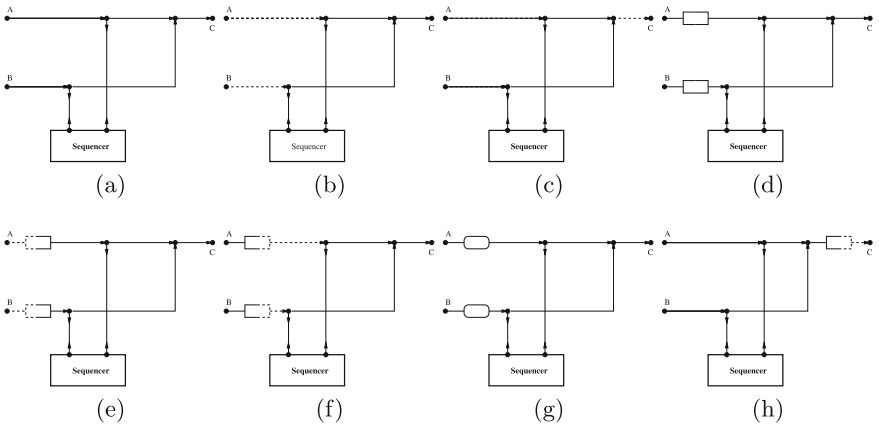


Fig. 5. Variants of alternating producers protocol

We can obtain new versions of our alternating producers and consumer example by attaching the ports of our producers and consumer to nodes A, B, and C of every connector in Fig. 5. All connectors in this figure share the same skeleton structure, based on a two-node version of a *sequencer* connector. Detailed description of the sequencer and these connectors is beyond the scope of this paper. What matters for our discussion is that there are at least these other 8

different concrete protocols each of which with its own properties, that can serve as a suitable solution for an alternating producers and consumer application. We can easily parameterize any of these connectors to scale up the number of their producers.

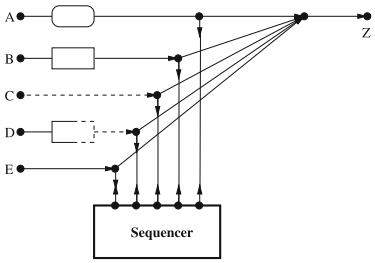


Fig. 6. Mix and match

Applications with many producers may indeed require somewhat different treatment of the output of some of their producers. It is trivial to mix-and-match the necessary interaction (sub-)protocols in Fig. 5, to tailor-make such a protocol, e.g., as in the example in Fig. 6. Such mix-and-match is generally unthinkable when protocols are expressed in terms of action-centric constructs of traditional models of concurrency.

8 Compilation

The examples in Sect. 7 exhibit the advantages of an interaction-centric model of concurrency that regards protocols as constraints. A high-level language like Reo that supports this form of protocol specification offers clear software engineering advantages (e.g., programmability, maintainability, verbatim-reusability, verifiability, etc.). However, as in constraint programming, it seems far less obvious that protocol specifications expressed in such a high-level language can be compiled into efficient and scalable executable code.

Recent results of our on-going work suggest that in time, sufficiently smart compilers for high-level protocol languages can generate executable code with better performance than hand-crafted code produced by programmers written in contemporary general-purpose languages with constructs of traditional models of concurrency. Superficially, our promising results may seem surprising and this claim, outlandish. Most of our results have already appeared in the literature [33, 35–43] and comprise the bulk of the work by Jongmans in his recently submitted PhD thesis [44]. Without getting into the technical details of how we obtained these results or the challenges that remain ahead, in this section, we summarize some of our results, and in the next section, describe a perspective on concurrent programming that "anticipates" our promising results and justifies the optimism of our claim.

Our compiler uses the constraint automata semantics of Reo [12]. It maps every node and every channel in a 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 that invoke the components. An external compiler for Java/C subsequently translates the full code base into a binary. Our Reo compiler currently applies a set of high-level optimization techniques on the intermediate constraint automata it produces. Some basic optimization methods identify groups of loosely- and tightly-coupled

small automata in order to improve scalability and strike a balance between low latency (sequentiality) and high throughput (parallelism) in the resulting executable code.

For some protocols, these optimizations already allow our compiler to generate code that can compete with code written by a competent programmer [39]. Figure 7 shows one of our most promising results. It shows the performance of three implementations of a k-producers-single-consumer protocol, for $k \in \{2^i \mid 2 \leq i \leq 9\}$: 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 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).

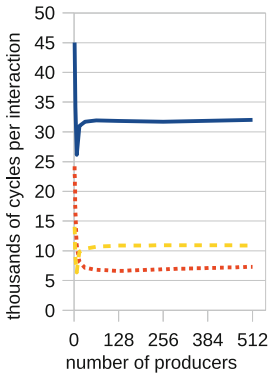


Fig. 7. Performance (Color figure online)

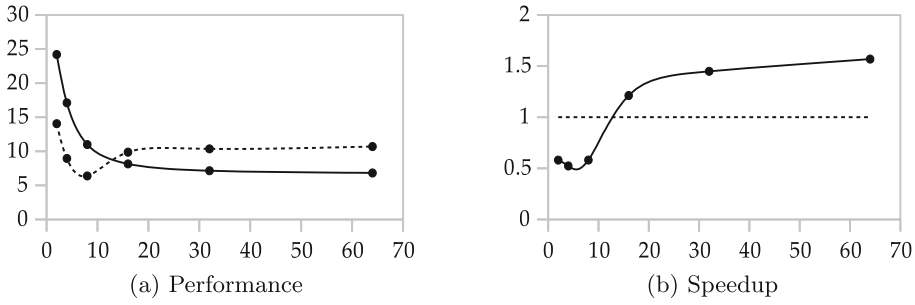


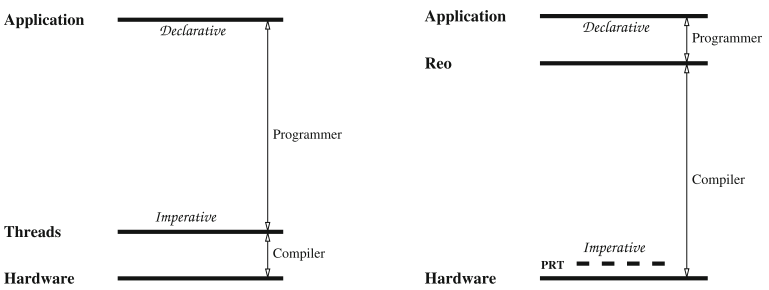
Fig. 8. Comparing hand-crafted (dashed) and Reo compiler generated (solid) protocol code: (a) thousands of CPU-cycles per protocol iteration vs. number of producers; (b) relative performance vs. number of producers.

In fact, this version of our compiler generates code that runs on the *Proto Runtime Toolkit* (PRT) [27,28]. PRT offers a run-time system for C code and a set of APIs. On its start-up, the PRT run-time system seizes control of the available cores from the operating system, thereby gaining full responsibility for scheduling instructions onto those cores. Software engineers use these cores through an API for managing PRT threads and a separate API for imposing custom scheduling policies. PRT-aware C code invokes the former API to instantiate units of parallelism, which the PRT run-time system subsequently schedules onto

cores, without interference by the operating system. Bypassing the operating system (and its rather heavy-weight scheduler) in this way, contributes to better performance. However, programming efficiently, directly at the level of PRT requires special skills. The PRT back-end of our Reo compiler shields programmers from PRT and its details, but reaps the benefits of improved performance that it provides, through a PRT API custom-made for Reo.

Figure 8 shows at a finer scale the performance and speedup of our Reo compiler generated code with that of a carefully optimized hand-written code using p-threads in C, for the above mentioned k-producers-single-consumer protocol. Figure 8(a) shows performance (in thousands of CPU-cycles per iteration of protocol, averaged over 10 runs) of the compiler generated (solid line) and the hand-crafted (dashed line) code as a function of the number of producers in this application. Figure 8(b) shows speedup of the compiler generated relative to the hand-crafted code as a function of number of producers.

These results show 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. The PhD thesis of Jongmans [44] discusses a number of formally sound high-level automata optimization techniques and contains extensive comparisons of our Reo compiler technology using the NAS Parallel Benchmarks [13]. His results demonstrate practical utility of verbatim reuse of protocols. They also show that in 37 % of cases our Reo compiler generated code is no worse than 10 % slower than the reference hand-crafted code of these benchmarks. In another 38 % of cases, our Reo compiler generated code is faster than their reference hand-crafted code. In the remaining 25 % of the cases, our Reo compiler generated code is between 10 % and 40 % slower. Some of these cases may improve by one or more of the many other high-level optimization techniques that we have not investigated yet. Nevertheless, these results offer preliminary evidence that programming concurrency protocols using high-level, interaction-centric constructs and abstractions can result in equally good—or better—performance as compared to hand-crafted code using conventional action-centric models of concurrency. Superficially, these results seem counter-intuitive. In the next section, we explain why, in fact, they are not.



(a) Action-centric (b) Interaction-centric

Fig. 9. Action-centric vs. interaction-centric protocol programming

9 Mind the Gap

Figure 9(a) shows three levels of abstraction of the protocol of a concurrent program. At the application level a protocol primarily expresses *what* it needs to accomplish, which essentially has a declarative nature. As an implementation in a conventional action-centric model of concurrency in a modern programming language, this protocol, for instance, turns into imperatives that control the scheduling of threads. Obviously, these imperatives must be refined into finer-grained imperatives of machine instructions before the application can actually execute on some hardware.

Figure 9(a), thus, shows that two transformations must take place before a specification of what a protocol needs to accomplish can actually run on some hardware: (1) translation by the programmer from the specification of *what* into the imperatives of *how*, e.g., expressed using the API of some threading library, and (2) translation by the compiler (of a conventional language) from the resulting threading API calls into executable machine instructions.

The distance between a pair of levels of abstraction in this figure suggests the complexity of abstraction/refinement transformation between them. One can, for instance, use the ratio of the number of lines of "code" that it takes to specify a protocol at each level of abstraction, and the mutual interdependence of these lines as a crude measure for this complexity. For example, the translation of an API call by the compiler of a conventional programming language into executable code may produce many lines of low-level code, but each such translation is quite straight-forward and the lines of code that result from two API calls essentially do not depend on each other (any more than the original two API calls did). As such, for instance, a C compiler contributes relatively little to the refinement into executable code of the protocol part of an application that is already expressed in terms of some threading API calls, as compared with the complexity of the refinement that the programmer performs, in order to transform the application-level specification of the protocol into precisely those specific threading API calls.

Programming of concurrency protocols is notoriously difficult precisely because the gap between the two levels of abstraction that specify what a protocol must accomplish and the imperatives that state how, represents a chasm of complexity. Programmers must navigate through this chasm essentially on their own, to produce *correct* imperative code. Additionally, programmers must also strive to manually make their correct code *efficient* too. And to reap the benefits of Gustafson's law, increasingly, programmers must also ensure that their correct, efficient code is *scalable* as well. These requirements make the manual translation of what a protocol must accomplish into how to do so imperatively, a very tall order that often frazzles even expert programmers. Because this translation substantially takes place in the mind of a programmer, even when it succeeds, it leaves no formal trace of its steps in the resulting code, from which a tool can subsequently reconstruct this translation or its inverse. Thus, the *intention* contained in *what* a protocol must achieve and the information about its translation into *how* its implementation does so are irrecoverably lost.

Imagine for a moment that instead of concurrency protocols we deal with our more familiar sequential programs, and consider an example of sorting an array of integers. A programmer may take this requirement (sort an array of integers) and produce a correct piece of code, written in a sequential language X. Assume that this code in fact implements a bubble-sort algorithm, perhaps because the programmer does not know any better sort algorithms. A compiler for X can do its best to generate optimized code for this program. However, can such a compiler look at all assignment, if-then-else, and for-loop constructs in its input, to divine from this jumble of source code that its programmer really intended to sort this array, and thus "compile" the bubble-sort algorithm that it finds in its input into the machine code for a quick-sort algorithm? Even if this transformation were theoretically possible, would it be desirable for a compiler to do so? After all, perhaps our programmer *did* actually know better and had a very good reason—unknowable to the compiler—to want a bubble-sort algorithm in this application.

Back to our concurrency protocol in Fig. 9(a), the fact that the programmer manually endeavors to translate what a protocol must accomplish into how to do so utilizing the low-level imperatives of an action-centric model, leaves relatively little wiggle room for the compiler to do significantly meaningful optimization *of the protocol*: following our sort analogy, above, it can optimize the implementation of each imperative, but it cannot compile its input imperatives of a "bubble-sort protocol" into the machine code for a "quick-sort" alternative protocol. Doing so requires a compiler to trace back the irrecoverable mental translation steps that the programmer took to produce its source code in the opposite direction, to divine the application level intention of the protocol; something of questionable desirability, even if theoretically possible.

Figure 9(b) shows the three levels of abstraction of the protocol of a concurrent program using an interaction-centric model of concurrency, such as Reo. The declarative, compositional constraint-programming style of protocol specification in Reo shrinks the gap between *what* a protocol must accomplish and its formal specification. As our examples demonstrate, this smaller gap makes it easier for a programmer to construct modular, verifiable, reusable, and scalable protocol specifications by composition. The programmer can now merely specify *what* (i.e., "sort") formally, instead of over-specifying *how* (i.e., "bubble-sort"), imperatively.[3] Shrinking the first gap also leaves much larger room in the second gap for a compiler to perform meaningful protocol optimization. In spite of its infancy, our compiler technology for Reo already demonstrates the practical feasibility of such meaningful optimization in our current results.

The specific version of the Reo compiler used in the benchmarks depicted in Figs. 7 and 8, generates C code that uses the API provided by the PRT system mentioned in Sect. 8. Thus, although this version of our Reo compiler does not generate object code that directly runs on the bare hardware, it indirectly does so assisted by the C compiler and the PRT run-time. We ignore this technical

[3] Of course, by adding extra "redundant" constraints, a programmer can also "specify" a bubble-sort, when and if desired.

detail in Fig. 9(b), because whether "compiler" in this figure designates a single Reo compiler or consists of a chain of automated tools does not change the point of our current discussion. However, for clarity, the dashed line in Fig. 9(b) shows the actual target level of abstraction of the concurrency constructs used in the C code generated by the version of our Reo compiler used in the above benchmarks: the PRT API. With respect to the other main levels of abstraction in this figure, PRT sits below the operating system, closer to the hardware, and offers concurrency constructs at a lower level of abstraction than that of operating system supported threading and scheduling facilities.

Fair Gain. A superficial reading of the "performance comparison" depicted in Figs. 7 and 8 may seem to reveal 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 in this case 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 the task assignment prevents the programmer from using lower-level constructs to directly hand-craft code similar to (or perhaps 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) develop tools that take p-threads level code written by a programmer and produce more optimized code; or (2) allow the programmer to directly code below the level of p-threads and OS.

Option 1 requires developing tools that can reconstruct the intentions behind the p-threads constructs used to encode a protocol (fragment); i.e., divine programmer's intention of "sort" from an imperative "bubble-sort" implementation code. Generally, this is impossible because the information about the mental transformation of *what* a protocol does into *how* it does it is irrecoverably lost. For instance, consider the piece of C code on the left. If its programmer 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 the programmer *additionally* intended the resulting array to have the same content, with the same random seed, in different executions (e.g., to reproduce bugs), a compiler cannot parallelize the loop: in that case, the order of generating random

numbers matters. Observe that from this code alone, neither a compiler *nor a human* can judiciously decide about loop parallelization; making such a decision requires *intention* information irrecoverable from this code.

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

Option 2, i.e., removing the artificial barrier of programming at the level of p-threads, is certainly possible. However, manually 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 the resulting code, in turn, sharply reduce the number of 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. Of course, the above drawbacks of producing programs at a level below p-threads and OS abstractions become moot if instead of a human programmer, a compiler performs this programming, starting with some high-level protocol specification.

While option 1, in principle, involves divining lost information, option 2 does not involve theoretical impossibilities; the difficulties in option 2 are "merely" technical and pragmatic. Our Reo compiler automates some of the technicalities involved in bypassing conventional concurrency constructs, making it more pragmatic to go from a high-level declarative specification of a *what* to a very efficient *how*-implementation below the level of p-threads or OS.

10 Concluding Remarks

Protocols constitute the most challenging aspect of concurrent applications. Specification of a protocol in action-centric models of concurrency invariably obscures what the protocol must achieve, because they lack mechanisms to forbid or even discourage dispersing constituent constructs of a protocol throughout an application software. Such dispersion intertwines protocol constructs with other data- and control-flow constructs of the application, which obfuscates the protocol, making it only an intangible by-product, implied by some sets of nebulous, logically-related-but-physically-scattered communication actions.

An increasingly important class of concurrent applications demand analysis, verification, reuse, composition, and scaling of protocols. Meeting the software engineering challenges of these applications requires definition and manipulation of protocols as proper mathematical objects, with composition and other operators to work with them. As a prime example of an interaction-centric model of concurrency, Reo can meet these challenges. Specification of protocols as declarative constraints in Reo makes them easier to manipulate and analyze directly, and makes it possible to compose protocols, scale, and reuse them verbatim.

The results of our ongoing work on compiling Reo suggest that, in addition to software engineering advantages, a high-level protocol language, such as Reo, can have advantages with respect to performance as well. Superficially, obtaining executable code that outperforms hand-crafted code, from the compiler of such a high-level protocol language seems counter-intuitive: one expects to pay the price of easier specification at a higher-level of abstraction, plus the software engineering benefits that it entails, by accepting a heavy penalty in performance. The perspective we described in this paper explains why avoiding such a performance penalty seems possible: compilers for such high-level languages can use formal information about what a protocol must achieve, to perform optimizations that compilers for lower-level languages cannot apply, simply because manual transformation by programmers irrecoverably loses such intention information.

References

1. Extensible Coordination Tools home page. http://reo.project.cwi.nl/cgibin/trac.cgi/reo/wiki/Tools
2. Reo home page. http://reo.project.cwi.nl
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
4. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: American Federation of Information Processing Societies: Proceedings of the AFIPS 1967 Spring Joint Computer Conference, 18–20 April 1967, Atlantic City, New Jersey, USA. AFIPS Conference Proceedings, vol. 30, pp. 483–485. AFIPS/ACM/Thomson Book Company, Washington D.C. (1967)
5. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
6. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Sci. Comput. Program. **55**(1–3), 3–52 (2005)
7. 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)
8. 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)
9. Arvind, Gostelow, K.P., Plouffe, W.: Indeterminancy, monitors, and dataflow. In: Rosen, S., Denning, P.J. (eds.) Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA, 16–18 November 1977, pp. 159–169. ACM (1977)
10. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 128–143. Springer, Heidelberg (2014)
11. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, New York (1990)
12. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)

13. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel benchmarks. IJHPCA **5**(3), 63–73 (1991)
14. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proceedings of SEFM 2006, pp. 3–12. IEEE (2006)
15. Benveniste, A., Caspi, P., Le Guernic, P., Halbwachs, N.: Data-flow synchronous languages. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803, pp. 1–45. Springer, Heidelberg (1994)
16. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Inf. Control **60**, 109–137 (1984)
17. Berry, G.: Esterel and Jazz: two synchronous languages for circuit design. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, p. 1. Springer, Heidelberg (1999)
18. Dean Brock, J., Ackerman, W.B.: Scenarios: a model of non-determinate computation. In: Díaz, J., Ramos, I. (eds.) Formalization of Programming Concepts. LNCS, vol. 107, pp. 252–259. Springer, Heidelberg (1981)
19. Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogenous systems. Int. J. Comput. Simul. **4**(2), 155–182 (1994)
20. Carbone, M., Yoshida, N., Honda, K.: Asynchronous session types: exceptions and multiparty interactions. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 187–212. Springer, Heidelberg (2009)
21. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: a declarative language for programming synchronous systems. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, 21–23 January 1987, pp. 178–188. ACM Press (1987)
22. Dennis, J.B., Gao, G.R.: An efficient pipelined dataflow processor architecture. In: Michael, G.A. (ed.) Proceedings Supercomputing 1988, Orlando, FL, USA, 12–17 November 1988, pp. 368–373. IEEE Computer Society (1988)
23. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer, Heidelberg (1999)
24. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: a declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987)
25. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. Pract. Exper. **40**(12), 1135–1160 (2010)
26. Gustafson, J.L.: Reevaluating Amdahl's law. Commun. ACM **31**(5), 532–533 (1988)
27. Halle, S.: A Study of Frameworks for Collectively Meeting the Productivity, Portability, and Adoptability Goals for Parallel Software. Ph.D. thesis, University of California, Santa Cruz (2011)
28. Halle, S., Cohen, A.: A mutable hardware abstraction to replace threads. In: Rajopadhye, S., Mills Strout, M. (eds.) LCPC 2011. LNCS, vol. 7146, pp. 185–202. Springer, Heidelberg (2013)
29. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21**(2), 289–300 (1993)
30. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)

31. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011)
32. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, 7–12 January 2008, pp. 273–284. ACM (2008)
33. Jongmans, S.-S., Arbab, F.: Global consensus through local synchronization: a formal basis for partially-distributed coordination. Sci. Comput. Program. **115–116**, 199–224 (2016)
34. Jongmans, S.-S., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
35. 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)
36. Jongmans, S.-S., Arbab, F.: Modularizing and specifying protocols among threads. In: Proceedings of PLACES 2012. EPTCS, vol. 109, pp. 34–45. CoRR (2013)
37. Jongmans, S.-S., Arbab, F.: Toward sequentializing overparallelized protocol code. In: Proceedings of ICE 2014, EPTCS, vol. 166, pp. 38–44. CoRR (2014)
38. 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)
39. Jongmans, S.-S., Halle, S., Arbab, F.: Reo: a dataflow inspired language for multicore. In: Proceedings of DFM 2013, pp. 42–50. IEEE (2014)
40. Jongmans, S.-S., Santini, F., Arbab, F.: Partially-distributed coordination with Reo. In: Proceedings of PDP 2014, pp. 697–706. IEEE (2014)
41. Jongmans, S.-S., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Orchestrating web services using Reo: from circuits and behaviors to automatically generated code. SOCA **8**(4), 277–297 (2014)
42. Jongmans, S.-S.T.Q., Arbab, F.: Can high throughput atone for high latency in compiler-generated protocol code? In: Dastani, M., Sirjani, M. (eds.) FSEN 2015. LNCS, vol. 9392, pp. 238–258. Springer, Heidelberg (2015)
43. Jongmans, S.-S.T.Q., Santini, F., Arbab, F.: Partially distributed coordination with Reo and constraint automata. SOCA **9**(3–4), 311–339 (2015)
44. Jongmans, S.-S.T.Q.: Automata-Theoretic Protocol Programming: Parallel Computation, Threads and Their Interaction, Optimized Compilation, [at a] High Level of Abstraction. Ph.D. thesis, Leiden University (2015, submitted)
45. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing, pp. 471–475. North Holland, Amsterdam (1974)
46. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: IFIP Congress, pp. 993–998 (1977)
47. Knight, T.: An architecture for mostly functional languages. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, pp. 105–112. ACM, New York (1986)
48. Liu, X., Xiong, Y., Lee, E.A.: The ptolemy II framework for visual languages. In: 2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), 5–7 September 2001, Stresa, Italy, p. 50. IEEE Computer Society (2001)
49. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)

50. Milner, R.: Elements of interaction - turing award lecture. Commun. ACM **36**(1), 78–89 (1993)
51. Ng, N., de Figueiredo Coutinho, J.G., Yoshida, N.: Protocols by default - safe MPI code generation based on session types. In: Franke, B. (ed.) CC 2015. LNCS, vol. 9031, pp. 212–232. Springer, Heidelberg (2015)
52. Ng, N., Yoshida, N.: Pabble: parameterised scribble for parallel programming. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, 12–14 February 2014, pp. 707–714. IEEE Computer Society (2014)
53. Sangiorgi, D., Walker, D.: Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York (2001)
54. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1995, pp. 204–213. ACM, New York (1995)
55. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Heidelberg (2014)