Modular Programming of Synchronization and Communication among Tasks in Parallel Programs

Modular Programming of Synchronization and Communication among Tasks in Parallel Programs

Bernie van Veen* and Sung-Shik Jongmans*†
*Department of Computer Science, Open University of the Netherlands
†Department of Computing, Imperial College London

***Abstract*—Implementing synchronization and communication among tasks in parallel programs is a major challenge. We present a high-level DSL geared toward this challenge, by generalizing the existing protocol language Reo from supporting only a compile-time/statically set number of tasks (unsuitable for parallel programming), to supporting also a run-time/dynamically set number of tasks. Our contribution comprises new syntax, a new compilation/execution approach, and experimental results. Most surprisingly, the new approach can outperform the existing approach, even though the new approach requires more work to be done at run-time.**

***Keywords*-DSL; synchronization; communication; coordination; interaction**

I. Introduction

A. Synchronization and Communication

Implementing synchronization and communication among tasks in parallel programs is a major challenge. Broadly, there are two strategies.

The first strategy is to use a language/library that *completely hides* the complexities of synchronization and communication (e.g., implicit/data parallelism [1]–[5]; algorithmic skeletons [6]–[8]; DSLs [9]–[11]). This strategy is characterized by two properties. First, getting synchronization and communication right *is not* the responsibility of the programmer, but of the implementor of the language/library (e.g., a pipe skeleton ensures safe communication of values between the parallel stages of a computation). Second, the language/library is typically less generally applicable (e.g., the parallel program needs to "fit" the algorithmic skeletons).

The second strategy is to use a language/library that *reduces* the complexities of synchronization and communication, without completely hiding them (e.g., transactional memory [12], [13]; coordination languages [14], [15]). In this second strategy, getting synchronization and communication right *is* the responsibility of the programmer, but the language/library makes it simpler and is typically more generally applicable.

This paper is about the design and implementation of languages/libraries that support the second strategy. We argue that such languages/libraries should be built upon the principle of *separation of concerns* [16].

B. Separation of Concerns

In the context of synchronization and communication, "separation of concerns" entails dividing a parallel program into syntactically separate *task modules* and *protocol modules*. Every task module encapsulates a task; every protocol module encapsulates synchronization and communication between those tasks. We use the qualifier "protocol" because synchronization and communication forces tasks to (inter)act only according to well-defined "rules of engagement" (= protocol); harmful interference among tasks, in particular, should be ruled out.

Examples of simple protocol modules are semaphores, barriers, and concurrent queues (channels). More complex protocol modules may be multiparty, stateful, and/or asymmetric. A basic example of such a protocol is the following:

Example 1. *<u>First</u> task A communicates a message to task C, <u>then</u> task B communicates a message to C.* □

The synchronization and communication required to enforce this protocol should ensure not only the fidelity of the message communications between the tasks (e.g., avoid data races), but also that the communication from A to C *strictly precedes* the communication from B to C.

Already in the early 1970s, Parnas attributed three general advantages to modularity [17]:

> "(1) *managerial*—development time should be shortened because separate groups would work on each module with little need for communication; (2) *product flexibility*—it should be possible to make drastic changes to one module without a need to change others; (3) *comprehensibility*—it should be possible to study the system one module at a time."

A fourth advantage is (4) *reusability*—it should be possible to straightforwardly "plug in" a module from a previous program into a new one.

But despite these advantages, and while separation of concerns has pervaded many software engineering practices, implementing synchronization and communication in parallel programs is not one of them. For instance, popular frameworks, as MPI and OpenMP, do not provide linguistic support to separate tasks from protocols. To implement Ex. 1

in MPI, the real communications need to be supplemented with an auxiliary barrier or communication, but MPI has no constructs to correlate the corresponding auxiliary function calls with the function calls for the real communications, as a single protocol implementation. As a result, the previous four advantages do not apply.

C. Contribution and Organization

Our longer-term research aim is:

To provide parallel programmers a language with high-level abstractions for synchronization and communication, built upon the principle of separation of concerns, to provide the advantages of modularity (1), (2), (3), (4), as described above.

In pursuit of this aim, one existing language has our particular interest: *Reo* [18]. Originally, Reo is a language for specification of protocols among components in component-based systems. However, it has qualities that make it attractive for our purpose as well: Reo provides high-level, graphical abstractions (protocols are specified as graphs, reminiscent of data-flow diagrams, but more expressive); it is intimately built upon the principle of separation of concerns (components in Reo are oblivious to the synchronization and communication between them); it has formal semantics and rigorous tool support (e.g., compilers for code generation [19]–[21]; model checkers for verification [22]–[24]). As a well-studied language, under research and development for over a decade, Reo seems exactly what we aim parallel programmers to provide. But, it is not.

A core assumption in Reo is that the subjects of synchronization and communication are known at compile-time. Although this is fine for Reo's original use cases, it is unreasonable in parallel programming, where (numbers of) tasks are typically set at run-time. Essentially, using Reo for parallel programming results in programs "able to use only a fixed number of processors", which are "bad programs" [25].

Thus, there exists a language—Reo—for defining synchronization and communication, built upon a principle—separation of concerns—that has a number of important advantages; this is exactly what we want. Yet, we cannot use this language in parallel programming, because it yields "bad programs". The main contribution of this paper is, therefore, a generalization of Reo to support parallel programming:

- We present new syntax that allows protocols specified in Reo to be parametrized in numbers of tasks.
- We present a new compilation/execution approach for the new syntax. This is challenging, as Reo's existing compilation/execution approach relies on compile-time knowledge of numbers of tasks.
- We present experimental results. Most surprisingly, the experimental results show that the new approach can outperform the existing approach, even though the new approach requires more work to be done at run-time.

```
public interface Outport { // implemented by OutportImpl
  void send(Object o); }
public interface Inport {  // implemented by InportImpl
  Object recv(); }
public interface Channel { // implemented by ChannelImpl
  void connect(Outport out, Inport in); }
```

Figure 1: Foster-Chandy model

```
public class Tasks {
  public static void a(Outport out) {
    Object o = ...; out.send(o); }
  public static void b(Inport y, Outport out) {
    Object o = ...;          // ^^ auxiliary
    y.recv(); // << auxiliary
    out.send(o); }
  public static void c(Inport in1, Inport in2, Outport x) {
    Object o1 = in1.recv();                  // auxiliary ^^
    x.send(0); // << auxiliary
    Object o2 = in2.recv();
    ...; }
  public static void main(String[] args) {
    Outport ao = new OutportImpl();
    Outport bo = new OutportImpl();
    Inport ci1 = new InportImpl();
    Inport ci2 = new InportImpl();
    new ChannelImpl().connect(ao, ci1);
    new ChannelImpl().connect(bo, ci2);
    // auxiliary:
    Outport x = new OutportImpl();
    Inport  y = new InportImpl();
    new ChannelImpl().connect(x, y);
    // tasks as threads:
    new Thread() { public void run() { a(ao); } }.start();
    new Thread() { public void run() { b(y, bo); } }.start();
    new Thread() { public void run() {
      c(ci1, ci2, x); } }.start(); } }
```

Figure 2: Example 1 (Foster-Chandy model)

Sect. II motivates the programming model. Sect. III summarizes Reo. Sect. IV presents the design of the new syntax and the new compilation/execution approach. Sect. V presents an implementation and experimental results.

II. Programming Model

Our starting point is the *Foster-Chandy model* for parallel programming [25], [26]. In this model, every parallel program consists of *tasks*, which execute concurrently. To synchronize/communicate with other tasks, every task has an interface consisting of *outports* and *inports*, enabling it to send and receive *messages*. An outport/inport pair can be connected by a *channel* with an unbounded buffer. Send operations on the inport are nonblocking, while receive operations on the outport are blocking (they complete only once a message becomes available). Fig. 1 shows three interfaces in Java to implement the Foster-Chandy model.

Example 2. *Fig. 2 shows a Java implementation of Ex. 1 (Sect. I), using the interfaces in Fig. 1 (Foster-Chandy model). Class* `Tasks` *defines four static methods, which implement (stubs for) tasks A, B, and C, and a main task. The main task creates outports, inports, and channels.* □

Example 2 illustrates two important points.

(i) It shows that the level of abstraction in Ex. 1, is higher than the level of abstraction supported by the Foster-

```java
public interface Connector { // implemented by ConnectorEx11
  void connect(Outport[] out, Inport[] in); }
```

Figure 3: Generalized Foster-Chandy model (cf. Fig. 1)

```java
public class Tasks {
  public static void a(Outport out) {
    Object o = ...; out.send(o); }
  public static void b(Outport out) {
    Object o = ...; out.send(o); }
  public static void c(Inport in1, Inport in2) {
    Object o1 = in1.recv(); Object o2 = in2.recv(); ...; }
  public static void main(String[] args) {
    Outport ao = new OutportImpl();
    Outport bo = new OutportImpl();
    Inport ci1 = new InportImpl();
    Inport ci2 = new InportImpl();
    new ConnectorEx11().connect({ao, bo}, {ci1, ci2});
    // tasks as threads:
    new Thread() { public void run() { a(ao); } }.start();
    new Thread() { public void run() { b(bo); } }.start();
    new Thread() { public void run() {
      c(ci1, ci2); } }.start(); } }
```

Figure 4: Example 1 (generalized Foster-Chandy model)

Chandy model: Ex. 1 orders the communications, which in the Foster-Chandy model can be enforced only with an *auxiliary communication* from C to B.

(ii) It shows that, although the Foster-Chandy model supports *modularity between tasks*, it does not support *modularity between task and protocol*. In particular, the synchronization and communication required to enforce the protocol is implemented *as part of* the three tasks; it is not encapsulated in a separate module and cannot, for instance, straightforwardly be reused.

To improve on these points, the Foster-Chandy model can be generalized (i) to support a higher level of abstraction and (ii) to support modularity between task and protocol. The idea is to allow *custom* n-ary synchronization/communication mediums among *arbitrary numbers* of outports and inports, called *connectors*, instead of allowing only channels (i.e., fixed binary mediums between a single outport and a single inport). Moreover, in the generalized model, not only receive operations are blocking, but also send operations.[1]

Fig. 3 shows an interface for a Java implementation of the generalized Foster-Chandy model; we also need interfaces `Outport` and `Inport` (Fig. 1), but with different implementations. While interface `Channel` (Fig. 1) has only one implementation, interface `Connector` has many, each of which *comprehensively* encapsulates *all* synchronization and communication required to enforce one protocol.

Example 3. *Fig. 4 shows a Java implementation of Ex. 1, using the interfaces in Fig. 3 (generalized Foster-Chandy model). Class* `ConnectorEx11` *(omitted) implements interface* `Connector`*; it comprehensively encapsulates all synchronization and communication.* □

[1]However, a connector with an internal buffer may immediately accept any message sent on an outport, and store it in its buffer (if the buffer is not yet full), thereby making the send operation essentially nonblocking.

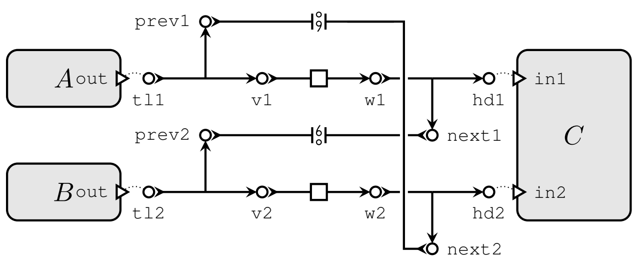


Figure 5: Example 1 (Reo; cf. Fig. 4)

Example 3 illustrates two points opposite to Ex. 2:

(i) There is no need for an auxiliary communication from C to B; the internals of `ConnectorEx11` ensure that a send operation on `bo` will not complete—sends are blocking in the generalized Foster-Chandy model—before a receive operation has completed on `ci1`.

(ii) All synchronization and communication required to enforce the protocol is encapsulated in a separate module: class `ConnectorEx11`. As such, it is provides the four advantages listed in Sect. I.

To implement programs that consist of $n > 1$ different protocols, n different connectors are needed. The advantages of the generalized Foster-Chandy model are also significant in this case: per-protocol encapsulation of synchronization and communication means that each of the n protocols can be implemented as a connector, in isolation. In contrast, in the basic Foster-Chandy model, at least n channels are needed, while the code for every protocol (i.e., usage patterns of those channels) is mixed not only with the task code, but also with code for other protocols.

Of course, it is possible to program `ConnectorEx11` manually. However, doing so places the complexities of implementing synchronization and communication, using rather low-level constructs (shared memory, locks, etc.), on the programmer. Instead, higher-level abstractions should be available, from which lower-level code can automatically be generated (and integrated with the rest of the program).

This is what Reo provides: Reo is a language to specify connectors (protocol modules) among "connectees" (task modules), each of which comprehensively encapsulates synchronization and communication to enforce a protocol, built on the generalized Foster-Chandy model. The connectors can subsequently be formally verified through model checking (e.g., to prove deadlock freedom or temporal logic properties), fully automatically [22]–[24]. Once everything is shown to be in order, the Reo compiler can be used to generate lower-level code.

III. Reo

A. Syntax and Informal Semantics

Example 4. *Fig. 5 shows a Reo implementation of Ex. 1. Reo has a graphical syntax. A* Reo diagram *consists of*

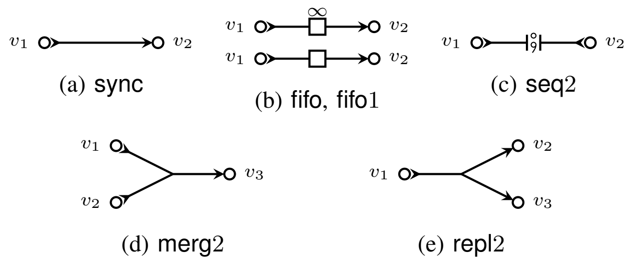


Figure 6: Example primitives

connectors (protocol modules; the graph in the middle) and connectees (task modules; the boxes on the sides), which are linked *(thin dotted lines) through outports and inports (outward and inward pointing triangles). The idea is that the graph gives an impression of the possible data-flows among the connectees; their internals are not defined in Reo, but in another language (e.g., the Java implementation of the tasks in Fig. 4). Shortly, we discuss Fig. 5 in more detail.* □

A connector (V, A) is a directed (hyper)graph of vertices V and (hyper)arcs A. Every arc $a \in A$ consists of a set of *tails*, denoted as $tl(a) \subseteq V$, a set of *heads*, denoted as $hd(a) \subseteq V$, and a *type* (graphically indicated by markings).

We use the following terminology. A connector is *primitive* if it consists of one arc; it is *composite* if it consists of more than one arc. A vertex is *public* if it has at most one incoming or outgoing arc; otherwise, it is *private*.

Example 5. *The connector in Fig. 5 is a composite. It has four public vertices.* □

Connectors can be *composed* using the union operator for graphs, denoted by $\oplus$: $(V_1, A_1) \oplus (V_2, A_2) = (V_1 \cup V_2, A_1 \cup A_2)$. Operator $\oplus$ gives rise to an alternative representation of a connector (V, A), namely as *a set of primitives* Γ. More precisely, let $prim(a) = (hd(a) \cup tl(a), \{a\})$ (i.e., $prim$ translates an arc to a corresponding primitive). Then, $\Gamma = \{prim(a) \mid a \in A\}$ implies $(V, A) = \bigoplus \Gamma$. Henceforth, we use only this alternative representation, because it allows for a more concise presentation of the key concepts.

We now briefly explain Reo's informal semantics.

Whenever a task performs a send/receive operation on one of its outports/inports, it effectively offers/accepts a message to/from the linked vertex in the connector. However, the operation is not immediately completed (i.e., a message is not directly sent/received): only whenever the connector is *ready* to handle the operation, *it* completes the operation. If the task should not block on the operation, this can be achieved in the connector by using asynchronous primitives, some of which are presented below (cf. Footnote 1).

The role of a connector is to transport messages between vertices, along its arcs, in response to messages being offered/accepted by tasks. Exactly when and where transportation of messages happens, is determined by the (global) semantics of the connector; this, in turn, is determined by the (local) semantics of its constituent primitives; this, in turn, is determined by the types of the arcs. Reo supports multiple arc types [18], a subset of which is shown in Fig. 6.

A sync primitive has synchronous semantics: in every execution step, a message synchronously flows from its tail to its head. A fifo primitive, in contrast, has asynchronous semantics: in every execution step, either a message flows from its tail into an internal unbounded buffer (always possible), or a message flows out of the buffer to its head (possible if the buffer is not empty). A fifo$n$ primitive (for some n) has similar semantics, except that its internal buffer is bounded by n: if the buffer is full, no new messages can enter. A seq2 primitive has two tails: in its first execution step, a message flows past the left tail (and is lost), and in its second execution step, a message flows past the right tail (and is lost). Graphically, the semicolon marking indicates "left and right" (i.e., which tail goes first). A merg$n$ primitive has n tails and one head: in every execution step, a message synchronously flows from one of its tails (nondeterministically selected) to its head. A repl$n$ primitive has one tail and n heads: in every execution step, a message synchronously flows from its tail to each of its heads. By definition, primitives repeat their execution steps infinitely.

In every instant, every constituent primitive of a connector has a number of possible *local execution steps*, according to its type. Before any of them does anything, all primitives first need to reach *consensus* about how to act collectively, to ensure that each of them acts individually in a way that is compatible with the others. For instance, a primitive must not offer a message to one of its heads, if that head is the tail of a fifo$n$ primitive whose buffer is already full. Once consensus is reached, all primitives act accordingly, and a *global execution step* of the connector emerges.

Example 6. *The (global) execution steps of the connector in Fig. 5 can be derived from the (local) execution steps of its constituent primitives as follows.*

Initially, the seq2 primitives accept a message only from their left tails. For the top seq2 primitive, this is `prev1`*; for the bottom one, it is* `next1` *(the semicolon is upside down).*

Since a message can flow past `prev1`*, and since a message can flow also past* `v1` *(because the buffer is initially empty), a message can flow past* `tl1`*. Thus, a send operation on the outport of A can immediately be completed. In fact, this is the only thing that initially can happen: for a send operation on the outport of B to complete, the message must flow past both* `prev2` *and* `v2`*, but the seq2 primitive connected to* `prev2` *does not (yet) accept messages from* `prev2`*; for a receive operation on an inport of C to complete, a message must be available in one of the buffers.*

Thus, initially, only a send operation on the outport of A can be completed. Subsequent (global) execution steps of the connector can similarly be derived (and yield Ex. 1). □

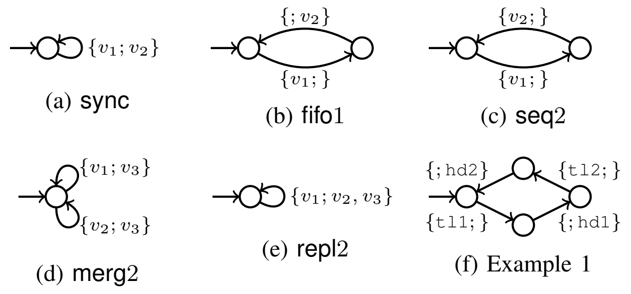


Figure 7: Example automata (cf. Fig. 6)

B. Formal Semantics and Compilation

Perhaps an obvious compilation approach for Reo is to translate every constituent primitive of a connector to a separate piece of lower-level code, and then run these pieces of code concurrently, while synchronizing their local execution steps through a consensus algorithm. Although conceptually simple (and close to the informal semantics), the consensus algorithm inflicts much overhead. To avoid this, Reo's most recent compilation approach works differently [20]. It is built on an automata-based formalization of Reo's informal semantics [27]; we explain this formalization next.

The idea is to represent the behavior of a connector Γ (i.e., a set of primitives) as a finite-state automaton $\alpha = [\![\Gamma]\!]$. States represent the connector's internal configurations, while transitions represent its (global) execution steps. Fig. 7 shows example automata. Every transition is labeled with a set of the vertices through which messages synchronously flow (in the execution step modeled by the transition). For instance, the transition in the automaton for a sync between vertices v_1 (tail) and v_2 (head) is labeled with $\{v_1; v_2\}$, where the semicolon separates tail(s) from head(s). In contrast, the automaton for a fifo1 has two transitions, to represent asynchrony. The automaton for a seq2 looks very similar; the only difference is the polarity of v_2. We remark that the transition labels in Fig. 7 are simplified relative to the transition labels used in the compiler (which have more information, notably about the content of messages), but these technicalities do not matter in the rest of this paper.

Automata can be composed using a (synchronous) *multiplication/product*, denoted by $\times/\prod$ [27]. Roughly, it consumes two automata α_1 and α_2 and produces a new automaton $\alpha_1 \times \alpha_2$ in which the transitions of α_1 and α_2 have been synchronized. This means that a transition of α_1 that involves *shared vertices* fires iff a transition of α_2 that involves *exactly the same shared vertices* fires; transitions of α_i that involve no shared vertices can fire independently. Let aut denote a function that maps primitives to automata. The automaton for a connector Γ is then computed as follows:

$$[\![\Gamma]\!] = \prod\{aut(\gamma) \mid \gamma \in \Gamma\} \tag{1}$$

To generate code for a connector Γ, first, the compiler looks up a "small automaton" $aut(\gamma)$, for every $\gamma \in \Gamma$, based on its type. Then, the compiler computes a "large automaton" by composing all the "small automata" (Eq. 1). Finally, the compiler generates a piece of lower-level code for a (sequential) state machine that reactively simulates $[\![\Gamma]\!]$.

At run-time, a generated state machine monitors the outports/inports linked to its vertices. Whenever a task performs a send/receive on one of its outports/inports, the state machine reacts by checking whether this operation enables a transition. If so, the state machine makes the transition, distributes messages accordingly among inports/outports, and completes all operations involved. If not, the state machine does nothing and awaits the next send or receive; all operations remain pending, and the sending/receiving tasks blocked.

The advantage of this compilation approach is that it completely avoids the need for consensus at run-time. In particular, by composing the "small automata" into a "large automaton", the compiler already statically computes all global execution steps that consist of mutually compatible local execution steps (as Reo's informal semantics demands). The reason this works, is because $\times$ is defined such that it synchronizes transitions of the "small automata" iff the corresponding local execution steps are compatible.

Reo is fully independent from both the host language in which code is generated and the platform on which it is run. For instance, tools exist that generate code for Reo connectors in C [19], Java [20], JavaScript [21], and Scala [28]. Due to this independence, Reo can in principle be used to abstract away from platform-dependent synchronization and communication constructs and runtimes, and enforce protocols among tasks across heterogeneous platforms.

C. Remark on Actor Languages

Actor languages/libraries (e.g., Erlang, Scala) may seem similar to Reo, but there are fundamental differences.

First, the provided level of abstraction differs. For instance, Ex. 1 can surely be implemented with actors, but it requires the programmer to manually add an auxiliary communication from C to B, to ensure that B does not send to C prematurely (cf. Ex. 2). With Reo, in contrast, the programmer can write protocols at a higher level of abstraction, without manually programming such auxiliary interactions (which become lower-level implementation details).

Second, the automata (Fig. 7) for the primitive connectors (Fig. 6) can be implemented as actors, but to implement their synchronous composition, an extra consensus algorithm is necessary; it is more complex than just running primitive actors in parallel. For instance, the pipeline composition of two sync channels should behave as a sync channel, but if we run two actors for the sync channels in a pipeline without extra provisions, their compound behavior is asynchronous.

IV. Design

A. Overview

As stated in Sect. I, using Reo for parallel programming results in programs "able to use only a fixed number of processors", which are "bad programs". Indeed, as Sect. III showed, Reo has no notation to *parametrize* diagrams in the number of connectees. This makes Reo inadequate for implementing parallel programs, in spite of providing separation of concerns and the four advantages in Sect. I.

The main contribution of this paper is a generalization of Reo to support parallel programming. To achieve this, the following three issues need to be addressed.

- First, new syntax to express parametrization needs to be developed. The problem is that Reo's graphical syntax does not easily lend itself to an intuitive *and* expressive extension for parametrization.
  We solve this problem with a new *textual syntax*.
- Next, the compilation approach needs to be extended with support for the new textual syntax, and notably, for parametrization. The problem is that the existing compilation/execution approach relies on the assumption that all primitives in a connector Γ are known at compile-time, to compute Eq. 1. This assumption fails to hold with parametrization: Γ may depend on the number of connectees, known only at run-time.
  We solve this problem with a new *parametrized compilation* approach, which splits the work to be done (computation of Eq. 1) into a compile-time share and a run-time share. What can be done at compile-time, is done at compile-time; only the work that depends on the number of connectees, is deferred to run-time.
- Finally, the run-time system needs to be extended with the capability of performing the remaining work. The problem is that this should be done with minimal overhead: by moving work from compile-time to run-time, we may reasonably expect performance will suffer, but the price we pay should be as low as possible.
  We solved this problem with a *parametrized execution* approach, which uses "just-in-time composition" to do only the work that is really necessary.

The rest of this section presents the textual syntax, parametrized compilation, and parametrized execution in detail.

B. Textual Syntax

We first describe the basic idea behind the textual syntax without parameters, and then add parametrization to it.

The basic idea is to, instead of drawing a connector as a graph, write down a list of its constituents.

Example 7. *Fig. 8 shows a textual implementation of Ex. 1 (cf. the graphical implementation in Fig. 5). It consists of three* connector definitions *(lines 1–5, 7–9, and 11–12) and one* main definition *(lines 14–15).*

```
1  ConnectorEx11a(tl1,tl2;hd1,hd2) =
2        Repl2(tl1;prev1,v1) mult Repl2(tl2;prev2,v2)
3    mult Fifo1(v1;w1)        mult Fifo1(v2;w2)
4    mult Repl2(w1;next1,hd1) mult Repl2(w2;next2,hd2)
5    mult Seq2(next1,prev2;)  mult Seq2(prev1,next2;)
6
7  ConnectorEx11b(tl1,tl2;hd1,hd2) =
8        X(tl1;prev1,next1,hd1) mult X(tl2;prev2,next2,hd2)
9    mult Seq2(next1,prev2;)  mult Seq2(prev1,next2;)
10
11 X(tl;prev,next,hd) =
12   Repl2(tl;prev,v) mult Fifo1(v;w) mult Repl2(w;next,hd)
13
14 main = ConnectorEx11a(aOut,bOut;cIn1,cIn2) among
15   Tasks.a(aOut) and Tasks.b(bOut) and Tasks.c(cIn1,cIn2)
```

Figure 8: Example 1 (textual; cf. Figs. 4 and 5)

```
1  ConnectorEx11N(tl[];hd[]) =
2    if (#tl == 1) {
3      Fifo1(tl[1];hd[1])
4    } else {
5           prod (i:1..#tl) X(tl[i];prev[i],next[i],hd[i])
6      mult prod (i:1..#tl-1) Seq2(next[i];prev[i+1])
7      mult Seq2(prev[1];next[#tl])
8    }
9
10 main(N) = ConnectorEx11N(out[1..N];in[1..N]) among
11   forall (i:1..N) Tasks.pro(out[i]) and Tasks.con(in[1..N])
```

Figure 9: Example 8 (textual; cf. Fig. 8)

Every connector definition consists of a signature *(e.g., line 1) and a* body *(e.g., lines 2–5). A signature consists of a* name *(*`ConnectorEx11a`*) and a list of* formal parameters *(*`tl1, tl2, hd1, hd2`*). A body lists the connector's constituents, separated by keyword* **`mult`**, *alluding to operator* $\times$*. A constituent is either the instantiated signature of a primitive (defined as part of the language, including the Reo types) or composite (defined manually). Lines 7–12 exemplify the latter, where instances of* `X` *are constituents of* `ConnectorEx11b`*. Signatures are instantiated with formal parameters (for public vertices) or* local variables *(for private vertices;* `prev1, v1, w1, next1`*, etc.). All local variables are bound to a unique vertex (implicitly created), statically scoped, and inaccessible from outside the definition. The main definition consists of lists of instantiated connector signatures and task signatures.* □

To extend the basic textual syntax with parametrization in the number of tasks, we add a number of constructs: *arrays*, *conditional expressions*, and *iteration expressions*.

Example 8. *Imagine a parametrized version of Ex. 1, where task C receives messages from N tasks instead of two. Fig. 9 shows a textual implementation of this protocol.*

In the signature of the connector definition, the square brackets indicate that formal parameter `tl` *contains an array of vertices (line 1); we stipulate that arrays are nonempty. The length of* `tl` *is denoted by* `#tl` *(lines 2, 5, and 6), and it determines* which particular *constituents an instance of the connector consists of, through conditional and iteration expressions. If* `tl` *consists of one vertex, there is exactly one constituent (line 3). If* `tl` *consists of more than* *one vertex, in contrast, the number of constituents depends on the length of* `tl` *(lines 4–8). This is implemented using iteration expressions (lines 5 and 6), each of which consists of three parts: the declaration of an iteration variable, a range, and a body. The idea is that the body is instantiated for every value in the range (by binding the iteration variable to that value), and that each of these instantiated bodies is "in-lined" into the parent expression. Here, the bodies contain only a single constituent, but in general, it can be an arbitrary expression.*

In the signature of the main definition, parameter `N` *is declared. This parameter is an input for the program, and used at run-time to spawn an appropriate number of tasks, and to create correspondingly sized connectors.* □

The intended workflow is, first, to draw a connector in the graphical syntax; doing so gives a good impression of the intended flows between vertices, and it offers a means of rapid prototyping. Then, translate the (nonparametrized) graphical syntax to (nonparametrized) textual syntax. Finally, parametrize the textual representation by adding arrays, conditions, and iterations.

C. Parametrized Compilation

The main advantage of the existing compilation approach is that it completely avoids the need for consensus at run-time, by composing the "small automata" into a "large automaton" at compile-time. To do this, however, the set of all primitives in connector Γ must be known at compile-time. This depends on the constituents in the connector's definition; this, in turn, depends on the instantiation of conditions and ranges; this, in turn, depends on array lengths; this, in turn, depends on the number of connectees; and this, problematically, is not known at compile-time.

There is no way around this problem. The best we can do, to avoid run-time consensus as much as possible, is perform at compile-time all composition work that does not depend on the number of connectees; all remaining work is deferred to run-time. We discuss the compile-time share in this subsection, and the run-time share in the next.

To compile a connector definition, the first step is to *flatten* its body: all (non-primitive) constituents that occur in the body are (recursively) expanded and in-lined. Local variables in-lined in this way first need to be renamed to ensure they have unique names (their exact names are immaterial, because their scope is local; only uniqueness matters). After flattening, the body contains only primitive constituents, some of which may be nested in conditional and/or iteration expressions.

Example 9. *Flattening* `ConnectorEx11b` *in Fig. 8 yields* `ConnectorEx11a`*, up to associativity and commutativity of* **`mult`** *(and renaming* `v` *and* `w` *before in-lining* `X`*).* □

The second step is to look up a "small automaton" for every constituent and compose as many of them as possible

```
public class ConnectorExN implements Connector {
  private List<Automaton> automata = new ArrayList<>();

  @Override
  public void connect(Outport[] out, Inport[] in) {
    // construct "medium automata":
    if (out.length == 1) {
      automata.add(new Automaton1()); }
    else {
      automata.add(new Automaton2(out.length))
      for (int i = 0; i < out.length) {
        automata.add(new Automaton3(i)); }
      for (int i = 0; i < out.length - 1) {
        automata.add(new Automaton4(i)); } }
    // link outports/inports to vertices:
    ...; }

  // state machine for Fifo1(tl[1];hd[1]), line 3:
  private class Automaton1 implements Automaton { ... }
  // state machine for Seq2(prev[1];next[#tl]), line 7:
  private class Automaton2 implements Automaton { ... }
  // state machine for X(tl[i];prev[i],next[i],hd[i]), line 5:
  private class Automaton3 implements Automaton { ... }
  // state machine for Seq2(next[i];prev[i+1]), line 6:
  private class Automaton4 implements Automaton { ... } }
```

Figure 10: Example 8 (Java; generated for Fig. 9)

into a number of "medium automata" (instead of into one "large automaton"). To this end, the flattened body is first *normalized* into a form where all "small automata" occur together (which is necessary to subsequently compose them). Generally, an expression is in normal form iff:

- From left to right (separated by **`mult`**), it consists of: first a section with *only* (primitive) constituents, then a section with *only* iteration expressions, and finally a section with *only* conditional expressions.
- Nested expressions (i.e., bodies of iterations; branches of conditionals) are in normal form.

Computing normal forms is computationally easy.

Example 10. *To normalize* `ConnectorEx11N` *in Fig. 9,* `X` *is first expanded and in-lined (the body of* `X` *is already in normal form), and line 7 is then moved up.* □

After normalization, every expression in the flattened body (starting from the whole body) is translated to lower-level code, according to the following rules:

- For every instantiated signature in the constituents section, a "small automaton" is looked up. These "small automata" are composed into a "medium automaton", and lower-level code for a state machine is generated.
- For every **`prod`** `<var>:<range> <body>` in the iterations section, lower-level code for iteration is generated (e.g., for-loop), whose body consists of the code recursively generated for `<body>`.
- For every **`if`** `<cond> <branch1> <branch2>` in the conditionals section, lower-level code for conditional is generated, whose branches consist of the code recursively generated for `<branch1>` and `<branch2>`.

In Java, for instance, the generated code constitutes an implementation of interface Connector in Fig. 3.

Example 11. *Fig. 10 shows a Java implementation of the protocol specified in Ex. 8, generated from the textual implementation in Fig. 9 (simplified to save space).*

For every "medium automaton", a class that implements an interface `Automaton` *is generated (lines 19–25; details omitted). Instances of these classes are constructed in method* `connect` *(lines 5–16).* □

We call this new compilation approach *parametrized*. It strictly generalizes the existing compilation approach, in this sense: for connector definitions without arrays, conditionals, and iterations, the two approaches coincide.

D. Parametrized Execution

Execution of the generated code at run-time (e.g., method `connect` in Fig. 10), when numbers of connectees (i.e., array lengths) are known, yields a list of state machines for "medium automata". What remains to be done, is the work to compose these "medium automata" into one "large automaton". There are two approaches to do this.

The naive approach is to compose them immediately after they are constructed, before the actual computations have started. We call this *ahead-of-time composition*. The advantage is that it is easy to implement; the disadvantage is that resources may be spent unnecessarily, which happens if the "large automaton" has states that are never actually reached (not because they are theoretically unreachable, but because some paths are never followed).

A better approach is to generate only the part of the state space of the "large automaton" that is actually reached, as the program is executed. We call this *just-in-time composition*. The idea is to initially compute only the initial state (formed as the tuple of the initial states in the "medium automata"), plus the initial state's outgoing transitions (formed by synchronizing the outgoing transitions of the initial states in the "medium automata", as prescribed by $\times$). Only once a transition out of the initial state fires, that transition's target state is "expanded" by computing its outgoing transitions (in the same way as for the initial state)—and so on.

V. Implementation

A. Tools

We implemented the design in Java, extending the existing Reo-to-Java tools [20], although we do not use any Java-specific features; the design can implemented in other languages equally well. The implementation consists of the following components (Figure 11): an API, a graph-to-text translator, a text-to-Java compiler, and a runtime system. They are all implemented as plug-ins for Eclipse, as an extension to existing Eclipse plug-ins for Reo development [http://reo.project.cwi.nl], including a graphical editor, animation engine, and model checker.

The API consists, essentially, only of interfaces `Outport` and `Inport` (Fig. 1). Using this API, programmers can

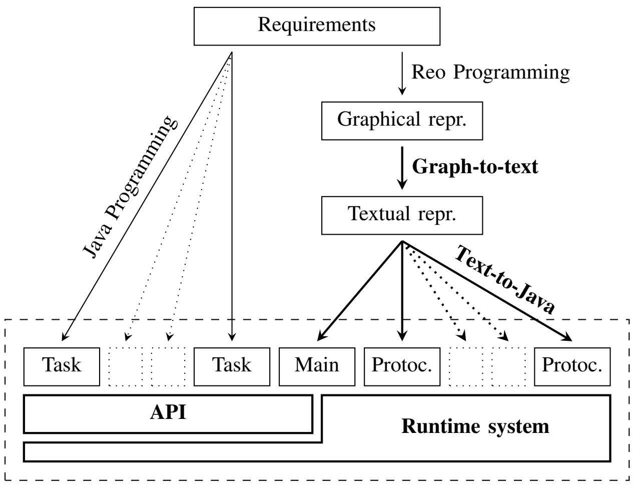


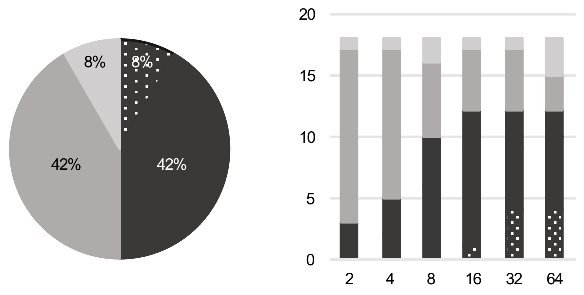
Figure 11: Bold font/thick lines indicate components of our implementation. Arrows represent workflow of the programmer. Rectangles indicate software artifacts. All software artifacts inside the dashed rectangle are Java sources (partially generated from Reo), which collectively constitute the executable parallel program (compiled by the Java compiler).

implement tasks as static methods in Java (Fig. 3). The graph-to-text translator consumes as input a Reo diagram, and it produces as output an equivalent textual representation (e.g., Fig. 5 to Fig. 8). The textual representation can then be parametrized (e.g., Fig. 8 to Fig. 9). The compiler consumes as input a textual representation, and produces as output lower-level Java code, as explained in Sect. IV. Finally, the runtime system provides an implementation of the API and some auxiliary classes. Notably, the runtime system supports both ahead-of-time composition and just-in-time composition of "medium automata" into a "large automaton"; this is set using a command-line flag.

B. Experiments: Connector Benchmarks

In our first series of experiments, we compared the performance of code generated for connectors using the existing compilation approach vs. the new one. In these experiments, thus, we concentrated on individual connectors instead of on full programs. We made a comprehensive selection of eighteen connectors, fully covering the major examples of parametrizable connectors in the Reo literature.

In every experiment, we first compiled the respective experimental connector for $N \in \{2, 4, 8, 16, 32, 64\}$ senders or receivers (depending on whether the connector is one-to-many, many-to-one, or many-to-many), with both the existing compiler and the new compiler. With the existing compiler, we needed to compile the connector six times, once for every value of N; with the new compiler, only one compilation was necessary. After compilation, we ran all generated code thrice for all N, on a machine with an



The pie chart summarizes all experimental results; the bar chart summarizes per N.

- Dark gray with dots: #experiments (y) in which new approach compiles successfully, whereas existing approach fails, for N senders/receivers (x)
- Dark gray: #experiments (y) in which new approach outperforms existing approach, for N senders/receivers (x)
- Medium gray: #experiments (y) in which existing approach outperforms new approach, up to one order of magnitude, for N senders/receivers (x)
- Light gray: #experiments (y) in which existing approach outperforms new approach, up to two orders of magnitude, for N senders/receivers (x)

Figure 12: Experimental results: Connectors (summary)

Intel i5-5300U processor (two cores[2] Hyper-Threading and Turbo Boost disabled), Windows 7 (64-bit), Oracle JDK 1.8 (max heap: 2 GB). For every run, we measured the number of global execution steps the connector (i.e., its generated code) made in four minutes. As we wanted to study the performance of the generated code, the tasks performed no computations; every task just tried to send and receive as often as possible.

Fig. 12 shows a summary of the experimental results; details are available elsewhere [29]. The overall trend to observe is that for smaller N, the existing approach generally outperforms the new approach, but for larger N, the new approach generally outperforms the existing approach. More in-depth analysis yields the following insights.

The two most interesting reasons why the existing approach can outperform the new approach:

1) The existing compiler does optimizations at compile-time, by simplifying transition labels (in a semantics-preserving way) [30]. This makes firing of *single transitions* at run-time (much) faster. These optimizations are also applicable in the new approach (but not yet implemented). For instance, in previous benchmarks with the existing compiler [30], speedups relative to unoptimized transition execution ranged from 1.2-fold for a single sync channel to 48.9-fold for a complex data-dependent connector (i.e., this optimization gets more effective as the size of the connector increases). As the complexity of applying this optimization is only linear in the size of the (unoptimized) transition label, we expect similar speedups in the new approach for protocols with loops (where run-time optimization costs are amortized over multiple iterations).
2) The existing compiler applies an optimization at compile-time, by analyzing the "large automaton" as a whole and manipulate its transition structure (in a semantics-preserving way) [19]. This makes firing of *sets of transitions* (much) faster. Contrasting the previous point, this optimization is *not* applicable in the new approach, because its application requires full knowledge of the "large automaton".

[2]The fact that the processor has only two cores, is not a problem given the aim of these experiments, as the generated code itself is not parallel.

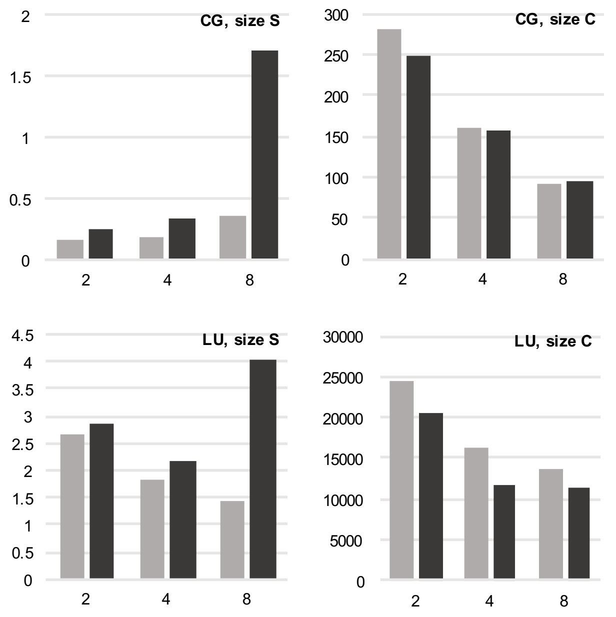
The connectors that suffer from point 1, are not fundamentally problematic: the transition-local optimizations applied in the existing approach, can be implemented in the new approach as well. The connectors that suffer from point 2, in contrast, *do* constitute an interesting next research challenge: we do not know yet if/how a transition-global optimization can be implemented in the new approach. This problem is important to solve, because it is the main reason why the existing approach outperforms the new approach up to two orders of magnitude in 8% of cases (red bins).

The most interesting reason why the new approach can outperform the existing approach is the very use of just-in-time composition. In particular, "large automata" that in theory have a number of states exponential in the number of "medium automata", can perfectly be handled in the new approach, because only a small part of such state spaces are actually reached at run-time, and because just-in-time composition computes only the part of the state space that is actually reached. In contrast, with ahead-of-time composition, the entire state space must necessarily be computed upfront, which the existing compiler cannot handle. Thus, in these cases, the existing approach failed, while the new approach worked fine.

In cases where the state-space is both exponentially large *and* each of those states may be reached at run-time in a sufficiently long run (unlike in the previous experiments), even then the new approach can have an advantage over the existing approach, when using a *bounded state cache*. The idea (not yet properly implemented) is to evict previously computed states from the cache if the cache is full (instead of saving them for eternity, as our runtime system currently does). The disadvantage is the possible need to recompute states that have already been computed, but also evicted, previously; the advantage is that arbitrarily large state spaces can be handled. We leave implementing such caches, and studying effective eviction policies, for future work.

C. Experiments: NAS Parallel Benchmarks

We compared the performance of hand-written code for a full program ("real" computations, plus synchronization and communication) vs. compiler-generated code using the new parametrized compilation approach. In these experiments, thus, we concentrated on full programs instead of on individual connectors. To this end, we took the Java reference implementation of the *NAS Parallel Benchmarks* (NPB) [31], which consists of seven programs: four kernels



CG is a kernel (master–slaves); LU is an application (master–slaves and pipeline).

- Dark gray: seconds of run time (y) of Reo-based programs, for N slaves (x)
- Light gray: seconds of run time (y) of original programs, for N slaves (x)

Figure 13: Experimental results: NPB (excerpt)

and three applications, derived from computational fluid dynamics software. In all programs, tasks are organized in a master–slaves structure; in one of the programs, additionally, the slaves are organized in a pipeline structure. We stripped the tasks in each of the programs from all synchronization and communication, and replaced it with (operations on) outports and inports.

In every experiment, we compiled the connectors in the respective program for $N \in \{2, 4, 8, 16, 32, 64\}$ slaves. After compilation, we ran both the original and its Reo-based variant, on a machine with four Intel E7-8890V3 processors (72 cores;[3] Hyper-Threading and Turbo Boost disabled), RedHat 7.3, OpenJDK 1.8 (max heap: 8 GB). NPB comes with a number of predefined workloads for all programs, of varying size (in increasing order: S, W, A, B, C). We measured the total run time.

Fig. 13 shows an excerpt of the experimental result; details are available elsewhere [29]. Our main findings:

1) The workloads of classes S and W are small; the overhead of the generated code dominates.
2) The workloads of classes A, B, and C are larger, and in those cases, the overhead of the generated code is amortized over the substantial work that the tasks need to do. As a result, the performance of the original programs and the Reo-based variants is comparable, for $N \in \{2, 4, 8\}$. This shows the new approach is also viable beyond synthetic benchmarks.
3) For $N \in \{16, 32, 64\}$, the Reo-based variants did not terminate within the time alloted, because the "large automaton" for the connector has some states with a number of transitions exponential in the number of slaves; just-in-time compilation does not help, because once such a state is reached, it is expanded, which requires computing its exponentially many transitions. This problem can be overcome by extending the new compiler with another existing optimization technique [32] (i.e., earlier experiments with NPB using the existing compiler, which *does* employ this optimization technique, showed comparable performance to the original programs [20]). This technique involves static analysis of the "small automata" (linear complexity), before they are composed into (a possibly exponentially) "large automaton". Based on this analysis (ahead-of-time), the set of "small automata" is partitioned (ahead-of-time), after which only automata in the same subset are composed (ahead-of-time *or just-in-time*). Using this technique, and with appropriate run-time support (of constant complexity, but non-zero), exponential growth can be avoided.

[3]Unlike in the connector benchmarks, the number of cores matters here.

VI. Conclusion

Separation of concerns is an important software engineering principle, with several well-documented advantages. However, when it comes to implementing synchronization and communication (= *protocols*) among *tasks*, concerns are typically not separated. To improve this, we aim to provide programmers a language with high-level abstractions for synchronization and communication, that naturally lets programmers separate tasks from protocols.

Reo is an existing language for specification of protocols among tasks, under research and development for over a decade, with a number of attractive qualities. Problematically, however, as Reo's original use cases did not demand it, Reo does not support specification of protocols parametric in the number of tasks. This makes Reo inadequate for parallel programming, despite its useful features and tools.

Reporting on a substantial initial effort, the main contribution of this paper is a generalization of Reo to support parallel programming. More specifically, we presented new syntax that allows specification of protocols that are parametric in the number of tasks; we presented the design and implementation of a new compilation/execution approach for the new syntax; we reported on experimental results. Most surprisingly, the new compilation/execution approach can outperform Reo's existing approach, even though the new approach requires more work to be done at run-time.

References

[1] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*. ACM, 2005, pp. 519–538.

[2] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *IJHPCA*, vol. 21, no. 3, pp. 291–312, 2007.

[3] G. L. Steele Jr., "Parallel Programming and Parallel Abstractions in Fortress," in *IEEE PACT*. IEEE Computer Society, 2005, p. 157.

[4] R. Chandra, *Parallel programming in openMP*. Morgan Kaufmann, 2001.

[5] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[6] M. Cole, "Algorithmic skeletons : a structured approach to the management of parallel computation," Ph.D. dissertation, University of Edinburgh, UK, 1988.

[7] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Softw., Pract. Exper.*, vol. 40, no. 12, pp. 1135–1160, 2010.

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating Code on Multi-cores with FastFlow," in *Euro-Par (2)*, ser. Lecture Notes in Computer Science, vol. 6853. Springer, 2011, pp. 170–181.

[9] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning," in *ICML*. Omnipress, 2011, pp. 609–616.

[10] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based PDE solvers," in *SC*. ACM, 2011, pp. 9:1–9:12.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*. ACM, 2013, pp. 519–530.

[12] N. Shavit and D. Touitou, "Software Transactional Memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[13] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *ISCA*. ACM, 1993, pp. 289–300.

[14] G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages," *Advances in Computers*, vol. 46, pp. 329–400, 1998.

[15] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[16] E. W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, ser. Texts and Monographs in Computer Science. Springer, 1982.

[17] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[18] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329–366, 2004.

[19] S. T. Q. Jongmans, S. Halle, and F. Arbab, "Automata-Based Optimization of Interaction Protocols for Scalable Multicore Platforms," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 8459. Springer, 2014, pp. 65–82.

[20] S. T. Q. Jongmans and F. Arbab, "PrDK: Protocol Programming with Automata," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 9636. Springer, 2016, pp. 547–552.

[21] M. Krauweel and S. T. Q. Jongmans, "Simpler Coordination of JavaScript Web Workers," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 10319. Springer, 2017, pp. 40–58.

[22] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz, "A Uniform Framework for Modeling and Verifying Components and Connectors," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 5521. Springer, 2009, pp. 247–267.

[23] N. Kokash, C. Krause, and E. P. de Vink, "Reo + mCRL2: A framework for model-checking dataflow in service compositions," *Formal Asp. Comput.*, vol. 24, no. 2, pp. 187–216, 2012.

[24] N. Kokash and F. Arbab, "Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools," *IEEE Trans. Services Computing*, vol. 6, no. 2, pp. 186–200, 2013.

[25] I. T. Foster, *Designing and building parallel programs - concepts and tools for parallel software engineering*. Addison-Wesley, 1995.

[26] I. T. Foster and K. M. Chandy, "Fortran M: A Language for Modular Parallel Programming," *J. Parallel Distrib. Comput.*, vol. 26, no. 1, pp. 24–35, 1995.

[27] C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten, "Modeling component connectors in Reo by constraint automata," *Sci. Comput. Program.*, vol. 61, no. 2, pp. 75–113, 2006.

[28] J. Proença, D. Clarke, E. P. de Vink, and F. Arbab, "Dreams: a framework for distributed synchronous coordination," in *SAC*. ACM, 2012, pp. 1510–1515.

[29] B. van Veen, "Improving scalability of concurrency in Reo," Master's thesis, Open University of the Netherlands, 2017.

[30] S. T. Q. Jongmans and F. Arbab, "Take Command of Your Constraints!" in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 9037. Springer, 2015, pp. 117–132.

[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. Weeratunga, "The Nas Parallel Benchmarks," *IJHPCA*, vol. 5, no. 3, pp. 63–73, 1991.

[32] S. T. Q. Jongmans, F. Santini, and F. Arbab, "Partially distributed coordination with Reo and constraint automata," *Service Oriented Computing and Applications*, vol. 9, no. 3-4, pp. 311–339, 2015.