Compiling Process Networks to Interaction Nets

Ian Mackie

Kahn process networks are a model of computation based on a collection of sequential, deterministic processes that communicate by sending messages through unbounded channels. They are well suited for modelling stream-based computations, but are in no way restricted to this application. Interaction nets are graph rewriting systems that have many interesting properties for implementation. In this paper we show how to encode process networks using interaction nets, where we model both networks and messages in the same framework.

1 Introduction

We relate two models of computation: Kahn Process Networks (KPNs) and Interaction Nets. Our aim is to encode process networks as Interaction Nets so that we can make use of implementations, and also to understand a different way of programming and computing within interaction nets. Specifically, we investigate encoding a KPN as a system of interaction nets, so from one perspective we can see this work as giving a compilation. Since there are many implementations of interaction nets this gives an implementation technique for process networks.

Kahn Process Networks [3] model computation using tokens travelling around a fixed network, and the structure of the network does not change during computation. Each node of the network transforms, depending on its function, tokens by consuming and creating new tokens. Transformation of tokens happens asynchronously and in parallel in the network: many tokens can be travelling around the network, and different nodes can be transforming different tokens at the same time. The communication channels, where the tokens flow, are buffered: the order of tokens is preserved, so tokens are queued waiting to be processed. Each processing element must block if inputs are not yet available, but they must not block to output. Only one process can write to each channel, and only one process can read each channel.

Interaction nets [5] on the other hand are a specific form of graph rewriting system. A program is represented as a network, and the net is rewritten to normal form by graph rewrite rules, called interaction rules. These rules act locally: no part of the graph can be copied or erased globally, so consequently one interaction cannot interfere with another interaction. Interaction rules can be applied asynchronously and in parallel; a feature shared with process networks. Each node of an interaction net must be connected by a single edge port-to-port to another node. Thus there are similarities between Interaction Nets and KPN.

Our goal is to simulate process networks in interaction nets. We achieve this by modelling both the fixed network and tokens using nodes in interaction nets. The firing of a transformation in the process network becomes an interaction rule (or several interaction rules). It is then just a convention that some nodes in the interaction net are fixed and others are travelling—in reality, we are just performing a rewrite rule. Not only does this give an implementation of data-flow networks with interaction nets, but it offers an interesting programming style that is frequently more natural (and often more efficient) than other ways. We see this as an application of interaction nets where the body of research on parallel implementation is becoming stronger.

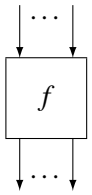
Preliminary Report. Final version to appear in:
TERMGRAPH 2016

Overview. The rest of this paper is structured as follows. In the next section we give some background material on process networks and set up the specific style of interaction nets that we are using, including some notation and conventions that we adopt. In Section 3 we build some example interaction systems that serve as building blocks for our compilation of data-flow networks. Section 4 gives an outline of the compilation process, and gives some examples of process networks in interaction nets. Section 5 gives an example of using the compilation, and also gives additional examples of programming with interaction nets in the style of process networks. Finally, we conclude in Section 6.

2 Background

Process networks. A Kahn Process Network [3, 4] is a directed graph $G = (V,E)$ with sequential processing units as the vertices V and unbounded buffers as the connecting edges E. A process, which is a sequential program that could be written in any language (for example C, Java, Algol, Haskell, etc.), communicates with another process if there is a connecting edge, called a channel, linking the two processes. A communication channel (stream or buffer) is a finite or infinite sequence of data elements (all the examples in this paper are numbers, but other data, included structured, is possible). We use a list notation $S = [x_1, x_2, ...]$, where the empty stream is denoted $[\,]$, when we need to write them down.

A process can be understood as a function mapping a collection of streams to another collection of streams. The following diagram illustrates the idea, where there are m inputs and n outputs:



Each process f is a sequential program that can read from input streams $S_1, \ldots, S_m$ to write to output streams $U_1, \ldots, U_n$ (we will assume $m, n > 0$) that can include the two primitives for communication:

- `send x on U`: this command sends a value x (we will assume all values are integers in this paper) along the channel U. The command is non-blocking, which is why the connecting edges are unbounded buffers, working as a FIFO queue.
- `x = wait(U)`: this command waits for a value on channel U, and assigns the value to the variable x. This command is blocking: a read on a channel must wait until the value is there, and only then consumes it.

If a process has only write instructions, then it may be the case that $m = 0$ is required. Such processes are said to be able to process on empty buffers, and this is typically needed to start the computation going. Here we will favour an alternative and initialise channels instead. If a process has repeated processing on empty buffers, then we can simulate this behaviour by creating a cycle by connecting an input from an output. The data on this extra channel can then be used to trigger the process. The process can then read this input, and just send it back around the loop to the input. In this way, all processes trigger output from input, but the price to pay is that we might need to initialise some of the channels with starting values.

Within a process, the `send` and `wait` instructions can be used in a number of different ways:

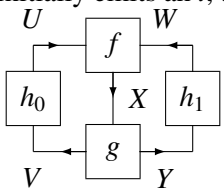
- A process does not have to read all of the inputs. Thus some channels can accumulate data indefinitely if a process is writing to the channel but no process is reading it.

- The order of input/output is not constrained in any way. A process can read from some of the channels in any order, then write to others. It can also interleave reading and writing.

A process has local variables, and thus an internal state. This is used to make the process behave differently on future inputs, and also to store past results. In the following, interaction nets will use different principal ports to mimic the different behaviour. The internal state of a process will be represented by the corresponding internal state of an interaction net node.

A process network is a collection of processes connected together. The edges are shared channels where communication can take place. The network of processes has a linearity constraint: only one writer per channel and one receiver per channel. A process can duplicate values on a channel (and write a value to two different channels) but each channel must be linear in this sense. A consequence of these conditions is that processes are deterministic. But they are also amenable to a high degree of concurrency: many processes can be active at one time, and each processor need only wait for all it's values before proceedings. For this reason process networks are a model of distributed computation. There are many variants of this model, for example synchronous processes. There are also many studies of properties of these networks, such as deadlock. Here we are interested in implementing them rather than studying any properties.

We give an example adapted from [3]. In this network f is a process that alternately receives input from the left/right and copies it to the output channel. g is a process that receives an input and alternately copies it left and right. h_i is a process that initially emits an i, then copies the input to the output channel.



The code for these processes is given by the following, where we just show, respectively, h_0 and f:

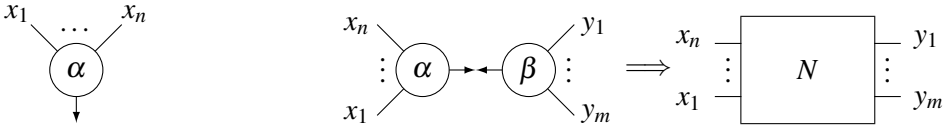
```
send 0 on U
repeat
  x = wait(V)
  send x on U
end
```

```
bool b = true
repeat
  if b then x = wait(U)
       else x = wait(W)
  send x on X
  b = not b
end
```

In this example, if we monitor the output of f, we get an infinite alternating sequence of 0 and 1. This process does not deadlock, and no synchronisation is needed for any of the individual processes.

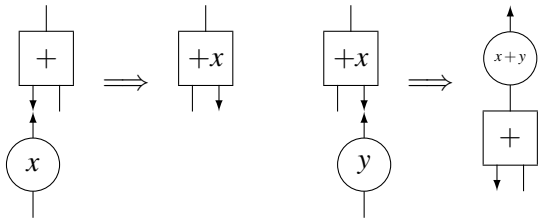
We refer the reader to the literature (see for example [3]) for a more detailed description of data-flow networks. Faustini [1] gives an operational semantics of process networks. There are many works implementing KPN (see for example [8] where they are encoded in Java), and there are many programming languages based on the principles. We give examples of these networks using interaction nets later.

Interaction nets. Analogous to term rewriting systems, we have a set of user-defined nodes (drawn as circles and squares in this paper), and a set of rewrite rules:

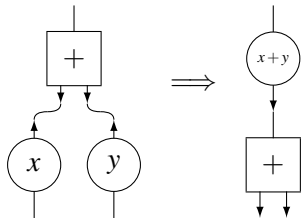


The set of rewrite rules is constrained: there is at most one rule for each pair of nodes; and each name in the diagram occurs exactly twice—once in the left, and once in the right. This means that the interface is preserved by reduction, and a consequence of this is that duplication and erasing must be done explicitly, but the rewriting system is one-step confluent by construction.

We extend interaction nets in two ways, and here we will present this informally. First, we allow nodes to hold values (just numbers are needed for the examples in this paper, but other base types can be included in a similar way). Consequently, rules can inspect and update these values. It is possible to do this and still preserve the same confluence properties by placing conditions on the rules. Next, we avoid introducing auxiliary nodes that are essentially required because arguments are taken one at a time (cf. Currying). We do this by allowing some nodes to have several principal ports, and the rewrite rule requires all to match. This offers no new computational power, but groups several interactions which is sometimes convenient. We illustrate all these ideas with an example. Consider the two rules which encode pairwise addition on lists of numbers:



we can combine these into a single rule in the following way:



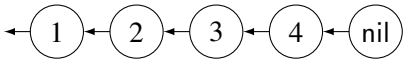
For the programs we write, all the properties of interaction nets are preserved, and this extension can be implemented in usual interaction nets. There are many implementations of interaction nets (see for example [2, 6]), including parallel ones. We will be building interaction net systems to simulate process networks, and for this to work we assume that the implementation is fair, by which we mean that no new reductions are done before older ones are completed. Most implementations respect this, so this avoids any starvation or live-lock issues that might arise.

3 Building blocks

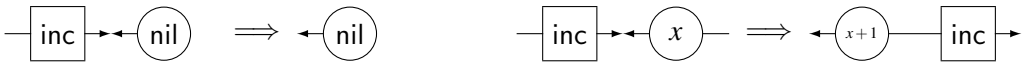
In addition to the examples for interaction nets given in the last section, we build some examples here that will be useful later for the compilation, specifically for the representation of the communication channels (buffers).

Lists and streams. Using two kinds of nodes, we can build lists of numbers:

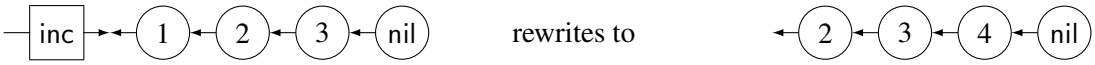
An example list with four elements would look like this:



Streams are just lists without nil. We can write simple operations over these lists/streams. For example, applying an operation to each element of a list (cf. map) can be done using the following two interaction rules, where we increment each element of the list:

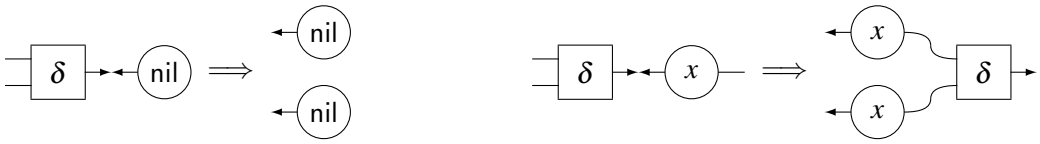


The following net rewrites using four interactions to reach a normal form:

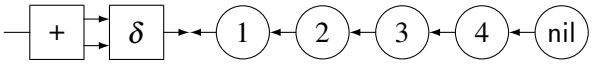


Other operations on lists/streams can be defined in a similar way.

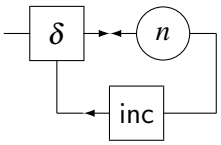
Duplicating streams. A very important operation, because of the linearity constraint, is the ability to explicitly copy lists and streams. With the introduction of a new node δ, we can define two interaction rules which will copy any list or stream in the following way:



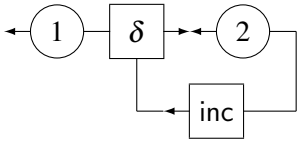
We can now put both of the above systems together to compute things like the following (that we leave as an exercise to the reader):



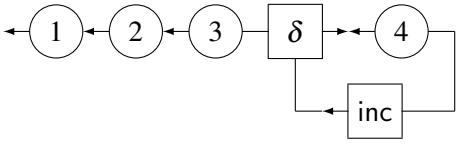
Using these components, we can generate an infinite list of integers, starting from a given number n with the following net:



We call this cyclic net ints(n). There are variants of this if we change inc to increment by 2, etc. Remark that this net is non-terminating. Two interactions are needed for each new number on the output stream (one to copy the number, and another to increment one of the copies). If we start with $n = 1$, and trace the execution (after two interactions each time) we have:



and then after four more interactions we have:



and so on. Thus we can generate cyclic nets, and simulate simple data-flow with interaction nets. This net generates an infinite stream of integers. This is an example of a process network, but uses our own interpretation on the nets: δ and inc are considered fixed and the numbers are interpreted as data items on a stream. In the next section we take these ideas further.

4 Compilation of process networks to nets

Our next task is to show how to translate any KPN into an interaction net. Not only does this highlight some connections between the formalisms, but it also provides an implementation of process networks. There are two possible ways to proceed:

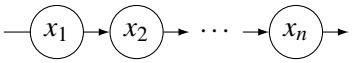
- We can define a set of KPN "combinators", by which we mean a finite set of processes that can be used to represent all other process networks (thus all computable functions). We then just need to show how to translate this fixed set of processes, together with the communication channels and net structure, to complete the compilation.
- We can give a general translation for each (user-programmed) process. This means that we need to fix the programming language used to define processes, and then give a translation of this language.

In this paper we just sketch the general translation, and then give some example interaction systems. We adopt a convention when writing interaction nets (that we have followed already in the previous examples): squares will represent processes, and circles will represent data items on channels (streams). The general form of the compilation is the following:

- Communication channels are encoded as interaction net streams, as defined in Section 3.
- Each process becomes a set of interaction net nodes that will simulate the functional behaviour of each process. We need to define the nodes and the rewrite rules for each process. Any internal state of a process becomes the internal state of an interaction node.
- The topological structure of a process network and an interaction net are identical, so this structure is just copied as part of the compilation.

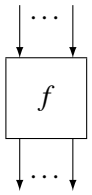
Below we give sufficient details of how process and channels are translated so that we are able to give some examples.

Channels. Each channel in a KPN is represented as an interaction net stream. If the channel is empty, then the stream is represented as an edge in the interaction net. Otherwise, if $S = [x_1, \ldots, x_n]$ then we build the following stream:

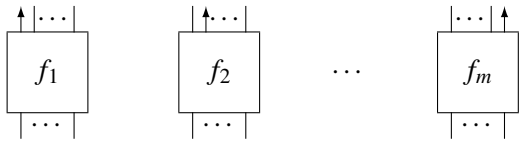


Initially, channels are usually empty at the start (so at the compilation). However, some exceptions to this rule will be used. In the compilation above, if we allow other data, then we need to translate that too. Here we just use numbers.

Processes. Each process f is represented by a collection of interaction net nodes. If f has m input streams and n output streams:



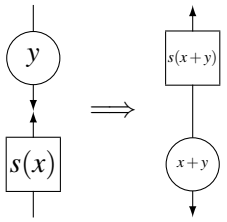
then this will be represented by a collection of m interaction nodes $f_1, \ldots, f_m$, where each f_i has the principal port at position i corresponding to the ith input stream, and each with $n+m-1$ auxiliary ports ($m-1$ input and n output):



For each node f_i we define the corresponding interaction rules that simulate the process. The interaction rule f_i with input on stream i becomes the process f_j (where $i = j$ is a possibility) and possibly outputs something on one of the output channels. The internal state is changed accordingly also.

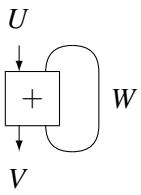
We note that a process network uses internal state in two very different ways: one to know which channel to read from, and the other to store values from past inputs. Interaction nets just use the second, because the choice of which channel to read from comes from the different node used (which gives the position of the principal port). Note also that if any channel is not read by the process, then there is no need to generate any rules for the corresponding node.

Example 1. *Consider a KPN that computes the running total from a channel. This can simply be implemented by a single process that stores internally, starting from 0, the accumulative sum of all the values read on the input channel. For each data item on the input channel, the total is updated, and the new total is written to the output. Following the ideas of the last section, we need just one node, and one interaction rule to represent this system:*



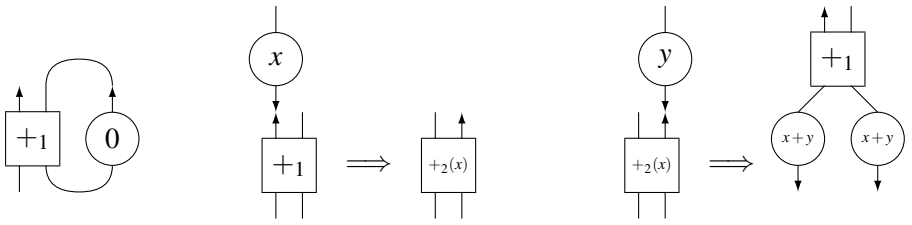
If we start with the node $s(0)$, we can generate the stream of accumulating totals read so far. The process keeps a history of the inputs (the sum of all the previous values).

Example 2. *Our next example is a variant on the previous one, where there is no internal state in the process. We use the network channels to keep the running total, which illustrates a general idea that we can frequently represent the internal state using additional channels. Consider the following process network:*




```
send 0 on W
repeat
  x = wait(U)
  w = wait(W)
  send (x+w) on V
  send (x+w) on W
end
```

This gets compiled into the following system. Since we need to send before read, we need to initialise the stream W with 0. The following is the initial net generated, and we give the two rules corresponding to the + process:



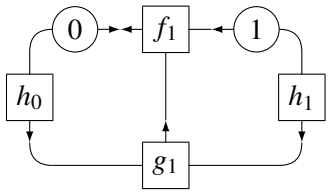
We end this section by stating a very general relation between KPN and the compiled interaction net.

Theorem 3 (Correctness)**.** *If a KPN reads or writes on a channel, then the corresponding interaction net can make the same move.*

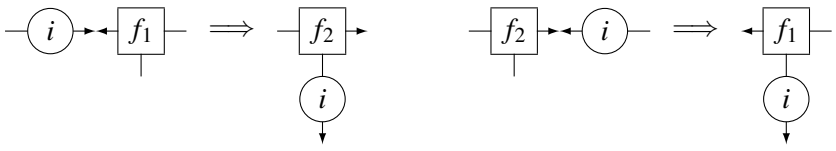
Because we mimic the functional behaviour of the KPN as a system of interaction nets, then this result is essentially obtained by construction. In a longer version of this paper we give the additional details to justify this. Finally, we remark that interaction nets are one-step confluent, therefore we get a completeness result also: it doesn't matter which way we evaluate a generated interaction net, it will always output the same thing as the KPN.

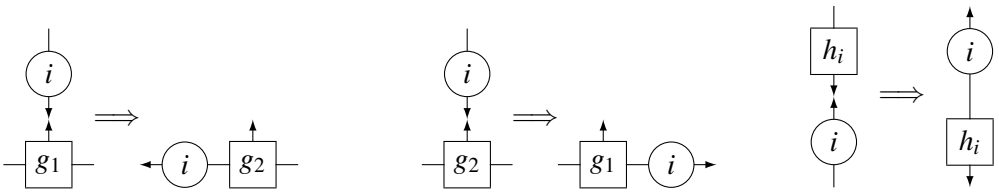
5 Representing process networks

Each process becomes a node in the interaction system, and we give rules that simulate the required behaviour. Channels become streams, and we represent streams of data as given previously. In this section we give another example of the compilation, and also give some examples of programming interaction nets directly in the style of process networks. The alternating bit process given in Section 2 can be represented by the following interaction net:

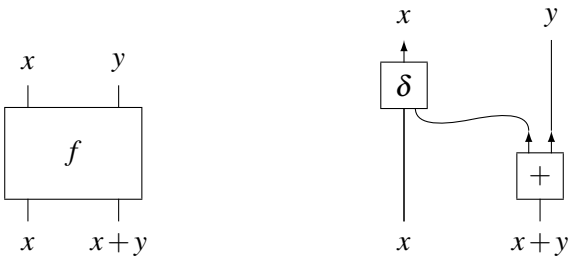


with the following six interaction rules (there are two rules for h_i that are identical):

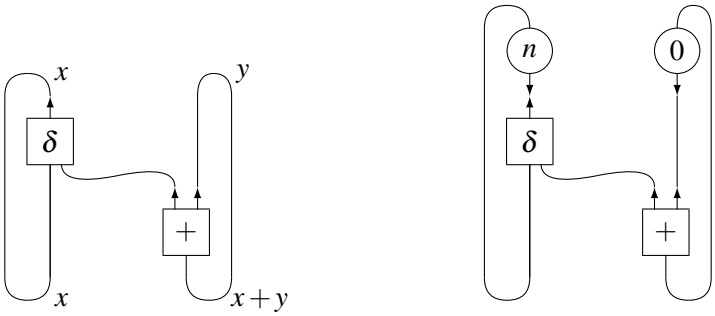




The choice of the principal ports directly encodes the order of reading on channels of the original process. We next show how to write process networks directly. Suppose we wish to build a net expressing the following input/output behaviour that we represent as a box f shown below. We can think of this as the function on pairs: $f = \lambda\langle x,y\rangle.\langle x,x+y\rangle)$. We can represent this using the building blocks we have already introduced, as shown below on the right:

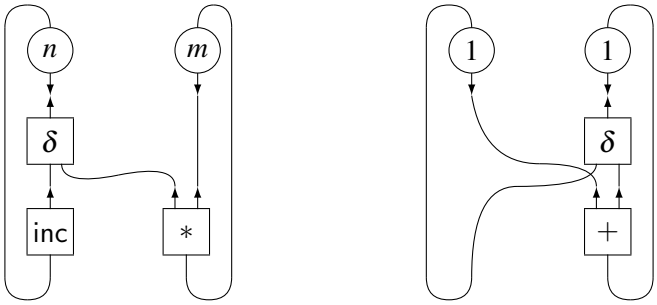


Connecting numbers to the top will give the required behaviour. We can iterate this function by building a feedback loop as shown below on the left. Finally, we can put some starting values in to complete the network, as shown below on the right:



If we look at the output of the + node, we will get n, $2n$, $3n$, etc. This network will compute a stream of multiples of n. We can also control the computation by restricting the iterations by introducing a counter. To do this we need to know when to stop, and how to extract the result. We can do this by adding in some synchronisation that can be encoded with standard ideas from interaction nets.

We give two final examples to illustrate how easily different nets can be built. These networks compute a stream of factorial numbers and a stream of Fibonacci numbers respectively:



6 Conclusion

Interaction nets give a very easy way of implementing data-flow networks. There are several parallel implementations of interaction nets now available, so this means that we can also take advantage of the natural parallelism that arises by writing programs in this way. Programming algorithms in interaction nets by encoding a process network can give a very efficient encoding of the algorithm: the example above for Fibonacci needs just three kinds of nodes and two rewrite rules to generate an infinite stream of Fibonacci numbers which is simpler and more efficient than any other way.

There are interesting extensions to the ideas presented here that are currently being investigated. For instance, is is possible to allow KPN to dynamically create new processes. This fits very naturally with interaction nets. Notions of second order networks (see for instance [7]) have been discussed, and

interesting relations again arise with past work on interaction nets. We hope to report on some of these ideas in a longer version of this paper.

References

[1] Antony A. Faustini (1982): *An Operational Semantics for Pure Dataflow*. In Mogens Nielsen & Erik Meineche Schmidt, editors: *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, *Lecture Notes in Computer Science* 140, Springer, pp. 212–224, doi:10.1007/BFb0012771.

[2] Abubakar Hassan, Ian Mackie & Shinya Sato (2014): *An Implementation Model for Interaction Nets*. In Aart Middeldorp & Femke van Raamsdonk, editors: *Proceedings 8th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2014, Vienna, Austria, July 13, 2014.*, *EPTCS* 183, pp. 66–80, doi:10.4204/EPTCS.183.5.

[3] Gilles Kahn (1974): *The Semantics of Simple Language for Parallel Programming*. In: *IFIP Congress*, pp. 471–475.

[4] Gilles Kahn & David B. MacQueen (1977): *Coroutines and Networks of Parallel Processes*. In: *IFIP Congress*, pp. 993–998.

[5] Yves Lafont (1990): *Interaction Nets*. In: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM Press, pp. 95–108, doi:10.1145/96709.96718.

[6] Ian Mackie & Shinya Sato (2015): *Parallel Evaluation of Interaction Nets: Case Studies and Experiments*. *ECEASST* 73, doi:10.14279/tuj.eceasst.73.1034.

[7] S. G. Matthews (1991): *Adding second order functions to Kahn data flow*. Technical Report, University of Warwick. Department of Computer Science.

[8] Thomas M. Parks & David Roberts (2003): *Distributed Process Networks in Java*. In: *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, Proceedings*, IEEE Computer Society, p. 138, doi:10.1109/IPDPS.2003.1213266.