

A Stream-Order Relaxed Execution Model for Asynchronous Stream Languages

Philip K.F. Hölzenspies
University of Twente
Department of EEMCS
P.O. Box 217
7500 AL Enschede
The Netherlands
p.k.f.holzenspies@utwente.nl

Jeyarajan Thiyyalingam Sven-Bodo Scholz Alex Shafarenko
University of Hertfordshire
Department of Computer Science
College Lane
Hatfield AL10 9AB
United Kingdom
T.Jeyarajan@herts.ac.uk

Abstract—In the context of typed asynchronous streaming networks, such as S-NET, the ordering of the stream is unimportant for stateless components of the network. Stateful components, however, do require the order to be preserved for correct execution. Allowing the stream to be evaluated out-of-order exposes concurrency, but introduces a need to locally restore this order for stateful components, which poses a local reordering problem. Reordering is further complicated by the multiplicity properly of components, i.e. the possibility of zero or more records being output in response to a single input record. This paper deals with this reordering problem and proposes a suitable execution model. The proposed model is capable of reordering records in the presence of stateful components and ensuring correct and maximum out-of-order execution.

I. INTRODUCTION

This paper deals with an execution model for languages describing the behavior of networks of asynchronous components and their orderly interconnection via typed streams. The purpose of asynchronous streaming network languages is to orthogonalise coordination and computation. The work in this paper will be explained in the terminology of one such language, namely S-NET [1].

A component is connected to the outside world by two typed streams: a single input stream and a single output stream. Data on these streams is organised as non-recursive, non-overlapping data items, called *records*. The components are asynchronous in the sense that they *may* start executing as soon as their input data is available. Upon execution, a component consumes a single record from its input stream and produces zero-or-more records in its output stream. This one-to-many concept is referred to as *multiplicity*. How many records are actually produced can depend on the record consumed, i.e. it is not necessarily predictable on the coordination level.

A useful advantage of orthogonalising coordination and computation is that it exposes concurrency. Components in S-NET work on a first-in-first-out basis, in the sense that when a record is consumed by a component, the component produces all the records in response to this record, before the next record is consumed. Since stateless components are not sensitive to the order of the stream, they can be executed in an out-of-order fashion. Such out-of-order execution of

components gives more scheduling freedoms, allowing ad hoc utilization maximization.

Stateful components, however, depend on the ordering of records in their input stream. An execution model for networks that contain stateful components must, therefore, guarantee the order in which records are presented to them (i.e. the ordering of the input stream of every stateful component). As a result, stateful components restrict concurrency.

The execution model presented in this paper tries to expose as much concurrency as possible, by allowing for as much out-of-order execution as possible. It does this, while still guaranteeing the (local) ordering of the streams flowing into stateful components.

The remainder of this paper is organised as follows. Section II describes network constructs (as used in S-NET) that are used to reason about networks in the remainder of this paper. The execution model is discussed in Section III. Section IV discusses related work before concluding the paper and making recommendations for future work in Section V.

II. NETWORK CONSTRUCTS

This section describes network constructs from S-NET. Networks are constructed from smaller networks (subnets) by means of combinators. Per combinator, the consequences for the flow direction, the ordering of streams and constraints on the execution model are discussed. A more detailed discussion of combinators can be found in [2]. Besides combinators, there are a few types of primitive networks, which will be discussed first.

A. Primitive network

A component (as mentioned in the introduction) is the simplest form of a network, i.e. a *primitive network*. The input stream of a network can be represented by a sequence of records, generally denoted with the first record on the left hand side. Thus, $\langle a, b, c, \dots \rangle$ denotes a stream in which record a is the first record. When this stream is the input stream for a primitive network N , the output stream will first contain all records resulting from the computation triggered by a ,

followed by all those from b , etc. This is denoted as

$$\llbracket N \rrbracket \langle a, b, \dots \rangle = \left\langle \underbrace{a_1^N, a_2^N, \dots, a_p^N}_{\text{response to } a}, \underbrace{b_1^N, b_2^N, \dots, b_q^N}_{\text{response to } b}, \dots \right\rangle$$

The number of records produced as a response to a record is defined in the component and, as such, non-deterministic on the network level. The same holds for the order of these records. This is because the network can not distinguish between individual records being produced by a component in response to a record being input. Thus, the order of a_i^N s in the above example is determined by the component.

There are three types of primitive networks in S-NET: boxes, filters and SynchroCells. Boxes are placeholders for code, written in some programming language (referred to as the box language). The behaviour of boxes is opaque in terms of execution time and multiplicity, but boxes are always understood to be stateless (no global variables are used, no side-effects introduced, etc.). Filters are simple operations on records (copying fields, copying entire records, deleting fields, etc.). They are written in S-NET, so their multiplicity is known. Because the representation of the contents of fields in records is specified by the box language, the execution time of filters depends on the execution time of the basic record operations in the box language. For the remainder of this paper, filters will be treated in the same way as boxes.

SynchroCells are the only stateful primitive networks in S-NET. They combine two or more records into a single record. SynchroCells can be understood to store n records, before producing the combined record. Which records to store is based on S-NET's type system [3], [4], which will not be treated in this paper. Instead, consider there to be n 'positions' in a SynchroCell and a function $\kappa : \text{record} \rightarrow \{1, \dots, n\}$ that projects every incoming record onto the position it corresponds to.

For every position i , only the first record a for which $\kappa(a) = i$ will be stored by the SynchroCell. All subsequent records corresponding to position i are passed through, i.e. produced unchanged on the output, immediately after being consumed from the input. When a record has been consumed for every position, a single record is produced on the output. After this, all records are passed through. The behaviour of the SynchroCell can be understood in terms of multiplicity, by observing that for all records consumed, at most one record is produced: The first $n - 1$ records that fill a previously unfilled position, have zero records produced in response. All other record consumptions result in one record being produced on the output.

B. Sequential composition

Given two networks N and M , the sequential composition (denoted $N \dots M$) is a network of which all input is streamed into N first and the output stream of N is the input stream of M . As such, the following equation holds for all N , M and a :

$$\llbracket N \dots M \rrbracket \langle a, \dots \rangle = \left\langle (a_1^N)_1^M, \dots, (a_1^N)_p^M, (a_2^N)_1^M, \dots \right\rangle$$

The sequential composition does not effect the order of records in the stream.

C. Parallel composition

Given two networks N and M , the parallel composition is a network of which the input stream is split up into two streams that are fed into N and M respectively, after which the output streams of N and M are merged. In S-NET the decision to let an incoming record stream either through N or through M is based on the type of the record and the input types of N and M . In more general terms, consider a function $\sigma : \text{record} \rightarrow \{\text{left}, \text{right}\}$ to determine how to split an input stream into the two input streams for N and M .

There are two variants of parallel composition: deterministic and non-deterministic. The difference between the two lies in the different definitions of the merger of the output streams. In deterministic parallel composition (denoted $N \parallel_\sigma M$), the merger of the output streams is defined to be order preserving, i.e. if record a streams into $N \parallel_\sigma M$ before record b , all responses to record a will stream out of the parallel composition before any of the responses to b . If, for example, a is routed to N and b is routed to M :

$$\llbracket N \parallel_\sigma M \rrbracket \langle a, b, \dots \rangle = \langle a_1^N, \dots, a_p^N, b_1^M, \dots, b_q^M, \dots \rangle$$

In non-deterministic parallel composition (denoted $N |_\sigma M$), the merger can arbitrarily interleave records from both output streams, but the order within each output stream is preserved. Thus, the following is a valid example of a non-deterministic merger:

$$\llbracket N |_\sigma M \rrbracket \langle a, b, \dots \rangle = \langle a_1^N, a_2^N, b_1^M, a_3^N, b_2^M, \dots \rangle$$

D. Recursion

Given a network N and a function $\gamma : \text{record} \rightarrow \{\text{left}, \text{right}\}$, a recursive network can be defined, as follows: Every record a for which $\gamma(a) = \text{right}$ is streamed out of the network. All other records are streamed through N . The output of N is again tested by γ to either be streamed out of the network or through (another instance¹ of) N , etc.

Again, there are two variants: deterministic (denoted N_γ^{**}) and non-deterministic (denoted N_γ^*). The deterministic variant preserves the order of the input, whereas the non-deterministic variant may not. Neither preserve the order of records produced *inside* the recursion. As an intuition, the recursion combinators can be described recursively as:

$$\begin{aligned} N_\gamma^* &\equiv (N \dots N_\gamma^*) |_\gamma Id \\ N_\gamma^{**} &\equiv (N \dots N_\gamma^{**}) \parallel_\gamma Id \end{aligned}$$

Note that only the $*$ combinator is recursive in itself, i.e. the $**$ variant is only deterministic at the outer most level. In the above, Id is the 'identity network' which, for every input record a , produces a and is order preserving.

¹When N contains only stateless components, this is irrelevant. When N contains stateful components, the state in one instance is independent of the state in another.

III. DATA FLOW TRAIN (DFT) EXECUTION MODEL

This section explains the Data Flow Train execution model. In contrast with a Communicating Processes approach, the driving element for concurrency in our approach is a record, i.e. primitive networks are not instantiated as threads, but rather every record in the stream is ‘handled’ by a thread. The motivational advantage of this approach is that records are no longer stalled by the records ahead of them in the stream.

Effectively, DFT decouples the semantic order (the order in which records occur in the input stream) from the temporal order (the order in which record handling threads reach a point in the network). For purely functional networks (without any stateful components), this approach provides a natural way of instantiating components based on ad-hoc demand. However, stateful components are defined on records in their semantic order. Records ‘arrive’ at components in *temporal order*, which is no longer the same as the semantic order in DFT. The semantic order, thus, needs to be (partially) reconstructed to preserve functional correctness of the execution.

The remainder of this section describes the structural representation of the input stream, the annotation of networks, the actual algorithms involved in implementing this approach and, finally, a concrete example.

A. Input representation

The global input stream is represented by a linked-list. Every node in this linked-list holds a single record and some administrative variables (see III-C) and it is associated with a thread². This representation offers two nice properties: Firstly, it allows for multiplicity and, secondly, inter-thread communication is singular and unidirectional (threads can only communicate with their direct successor).

Using a numbering of nodes in the stream, analogous to data communication networks [], is infeasible because of multiplicity. Any component may respond to any node with an unbounded multiplicity. Any finite numbering scheme of nodes has arbitrary and (compile-time) unpredictable limitations.

The semantic order is actually already contained in the representation of the input list. Since records take different routes through an S-NET network at run-time, the local order of records does not only rely on which record is a global predecessor of which other record, but also on whether a predecessor can actually reach the same location. Given a partial order on the components of a network, it is possible at any given time to deduce whether any predecessors of a node can still reach the node’s current position. Towards this, we propose the following enumeration scheme for S-NET networks.

B. Enumeration of the network

When a network consists of a single primitive network, the index of this primitive network is void. The following

²Actually, every thread is associated with *at least* one of these nodes. Letting a single thread handle more nodes reduces the amount of concurrency. When using threading models with high overhead, however, multiple *consecutive* nodes can be associated with a single thread. The remainder of this paper assumes one node per thread.

observations and rules constructively define indices for all possible networks.

- 1) Any non-primitive network can be divided into a number of subnets (N_i).
- 2) In any sequential composition $N_1 .. N_2 \cdots N_n$, every subnet N_i is numbered i .
- 3) (Hierarchy) All subnets *in* a subnet are similarly enumerated. For example, $[i, j]$ denotes the j^{th} subnet within the i^{th} subnet.

These indices are ordered lexicographically, i.e. (for all $i, j, k > 0$) $i < [i, j] < [i, j, k]$ and if $i < j$ then $[i, k] < j$. It should be clear that $i = [i, 0] = [i, 0, 0] = \dots$, but trailing zeroes are omitted. Because we only consider directed *acyclic* networks, indices of any two subnets can be compared to determine the reachability of one subnet from the other.

Definition. (Reachability) Given two subnets, N_i and N_j , N_i is said to be *reachable* from N_j (denoted $N_i \succ N_j$), iff $j < i$.

Syntactic note: The reachability operator can be used in both directions, i.e. $N_j \prec N_i \equiv N_i \succ N_j$. As added shorthand, the notation $N_i \succeq N_j$ is used to signify that either $N_i \succ N_j$, or $i = j$. This operator can also be used in both directions.

- 4) For a subnet $N_i \equiv M_\gamma^*$, every instance j of M is numbered as $[i, j]$. The lexicographic ordering of indices reflects the fact that records from later instances cannot flow into earlier instances.

Given a parallel composition of two subnets N_i and N_j , neither is reachable from the other. Since indices are used to deduce reachability, there should be no ordering of the indices of these subnets. In other words: The enumeration scheme should reflect the fact that no records from N_i can reach N_j and vice versa, i.e. it should hold that neither $N_i \succ N_j$ nor $N_i \prec N_j$.

To this end we introduce \mathcal{P} -numerals. Given any two integers n and m , it holds that $\mathcal{P}_n = \mathcal{P}_m \leftrightarrow n = m$ and when $n \neq m$ neither $\mathcal{P}_n < \mathcal{P}_m$, nor $\mathcal{P}_n > \mathcal{P}_m$. With these \mathcal{P} -numerals, parallel compositions can unambiguously be indexed.

- 5) In any parallel composition $N_1 | N_2 | \dots | N_n$, every subnet N_i is numbered \mathcal{P}_i .

Given the above enumeration scheme, any component within an S-NET can be assigned a unique index. The partial order of these indices provides a way to decide reachability of one component from any other.

Because indices only have a partial order, the minimum of two indices—in the conventional sense, where $\min(i, j) \in \{i, j\}$ —can not be defined. Instead, the minimum of two indices is defined as the largest common predecessor. It still holds that $\min(i, j) \leq i, j$ and $i \leq j \rightarrow i = \min(i, j)$. For example, $\min([1, \mathcal{P}_1, 4], [1, \mathcal{P}_1, 3]) = [1, \mathcal{P}_1, 3]$ and $\min([1, \mathcal{P}_2, 1], [1, \mathcal{P}_1, 4]) = [1]$.

C. Algorithm

A record streaming into the network corresponds to a node being attached to the end of the linked list (see III-A) and a thread being started. In an approach where network

components are threads, components consume records from their inputs. Since the DFT model dictates threads for records, stateless components can simply be seen as functions that need to be called by threads on their respective records.

Multiplicity needs some special attention. When a function is applied to a single record and results in a few (more than one) records, new nodes are inserted in the linked list by the thread calling the function for all but the first resulting records. They are inserted between the calling thread's own node and its successor. For every inserted node a new thread is started, after which the new node is no longer handled by the calling thread. The first result simply overwrites the record that was previously stored at the calling thread's node (which is also what happens when a function returns only one record). When the result of the function is zero records, the calling thread's node should be deleted. However, this node has a predecessor in the linked list, which is owned by another thread. Instead of deleting a node, therefore, the calling thread replaces its node with an indirection to its successor and then dies. The thread handling the node's predecessor may remove this indirection later on.

SynchroCells are not normal functions that can simply be called by individual threads, because they are stateful, i.e. each SynchroCell has a unique store associated with it. Considering that threads arrive at a SynchroCell in temporal order, but may only alter the SynchroCell's state in semantic order, a thread arriving at a SynchroCell must be suspended until it can be decided that it is the next in semantic order. Because the semantic order is preserved in the linked list, it suffices to conclude that none of the thread's node's predecessors will ever arrive at the SynchroCell. This is where the network indices come into play.

Every node in the linked list has a variable `index` which holds the node's current position in the network (as described in III-B). Another variable, `min`, contains the minimum of the indices of all the node's predecessors. More precisely, it holds the *propagated* index minimum: When a node progresses from one component to the next, its `index` is updated accordingly. If this update changes `min(index, min)`, the node sends this changed minimum to its successor (pointed to by the `next` variable). The `min` variable is the only shared variable between threads. This sharing is restricted, in that a node's `min` variable is read-only for that node and write-only for the node's immediate predecessor.

There are three primitive operations defined on the shared variables: `read(v)`, which reads the value stored in v non-destructively (v is unchanged), `consume(v)`, which reads the value stored in v destructively, and `produce(v,x)`, which writes value x into v destructively. Of these operations, `read()` and `consume()` are assumed to be blocking (when v is empty), whereas `produce()` is assumed to be non-blocking. Two consequences of this index administration are crucial to locally reorder nodes:

- 1) At any point in time, the values of all `min` variables are weakly decreasing through the linked list.
- 2) The value of any node's `min` variable weakly increases over time.

The procedure `Progress` performs this administration.

Procedure `Progress(c,i)`

```

1  $m \leftarrow \text{read}(c.\text{min})$ 
2  $m' \leftarrow \min(i, m)$ 
3 if  $m \neq m'$  then produce( $c.\text{next}.\text{min}, m'$ )
4  $c.\text{index} \leftarrow i$ 

```

Procedure `HitSyncCell(s,κ,n)`

```

1 repeat
2    $\text{suspended} \leftarrow \text{true}$ 
3    $m \leftarrow \text{consume}(n.\text{min})$ ; ▷ blocking
4    $q \leftarrow s.\text{queue}(\kappa(n)) \setminus \{n\}$ ; ▷ start critical section
5   if  $q.\text{min} < m$  then
6      $s.\text{queue}(\kappa(n)) \leftarrow \{n\}$ 
7     foreach  $n' : q$  do produce( $n.\text{min}, q.\text{min}$ )
8     if  $s.\text{index} = m$  then Sync( $s, \kappa, n$ )
9   else if  $q.\text{min} = m$  then
10     $s.\text{queue}(\kappa(n)) \leftarrow q \cup \{n\}$ 
11  else
12     $\text{suspended} \leftarrow \text{false}$ ; ▷ end critical section
13 until  $\neg \text{suspended}$ 

```

When a thread, with node n , arrives at SynchroCell s , it calls the procedure `HitSyncCell`. It is initially assumed (line 2) that the calling thread will be suspended. Every position of the SynchroCell (recall Section II-A) has a queue associated with it. The nodes of all suspended threads are stored in their corresponding queues (determined by κ). When $n.\text{min}$ exceeds the minima of the nodes in the corresponding queue (line 5), it may be concluded that the queued nodes are successors of n (because minima are weakly decreasing through the linked list). In this case, all previously queued nodes can be released (the SynchroCell passes through nodes for positions that are already filled) and n can be queued (lines 6–8). When $n.\text{min}$ is equal to the minima of the nodes in the corresponding queue (line 9), n is queued. The only other case is that $n.\text{min} < q.\text{min}$, which means that a predecessor of n is already on the queue and n can be passed through (line 12).

After line 3, $n.\text{min}$ is empty. After a thread has queued its node (and has thus remained suspended), the next iteration of the loop will be blocked by the consumption of the variable. When nothing arrives at a SynchroCell, but the predecessor of one of the queued nodes produces a new minimum index, that queued node will be unblocked and proceed to free the other nodes in its queue (because it was the first to obtain a higher value for its `min` variable and is thus their predecessor). Nodes are released from their queue with the value for `min` that they *had when they were queued* (line 7), so that the weakly decreasing nature of minima in the linked list is unchanged.

When node n is the first for which all predecessors have passed the SynchroCell (line 8), it is chosen for its corresponding position in the SynchroCell. If this was the last position to be filled, the node's record is overwritten with the combination of the records of all positions. If there are unfilled positions, the node is replaced with an indirection and the thread is killed.

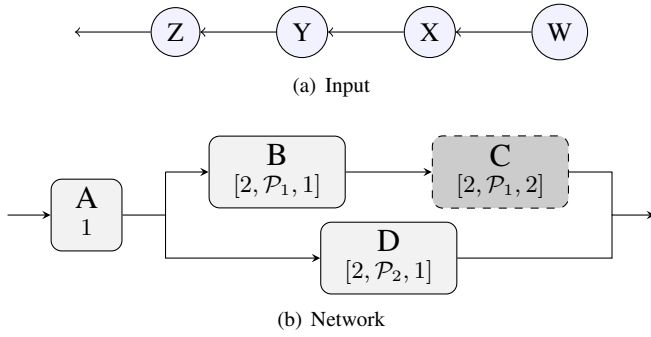


Fig. 1. Example scenario

| node | index | minimum |
|------|-------------------------|-------------------------|
| W | $[2, \mathcal{P}_2, 1]$ | $[3]$ |
| X | $[1]$ | $[2, \mathcal{P}_2, 1]$ |
| Y | $[2, \mathcal{P}_1, 1]$ | $[1]$ |
| Z | $[1]$ | $[1]$ |

TABLE I
POSITIONS OF THE NODES IN THE NETWORK

For brevity, the procedure `Sync`, that implements this, is not described in detail.

Lines 4–12 denote the critical section for the queue. This is a slightly more fine-grained control than making the loop body a critical section for the entire `SynchroCell`. Threads working on different queues do not block each other.

D. Example

Consider the example shown in Figure 1. The network (1(b)) is made up of four elements, labelled A through D . Under the element labels, the respective indices of the elements are shown. The input is represented by a linked list (1(a)) which contains four nodes, where the head of the list is W , followed by X , Y and Z , respectively.

Suppose that, in the example, C is a `SynchroCell` that should ultimately join together the records from nodes X and Y . Furthermore, assume that W will flow through the lower branch of the parallel composition, X , Y and Z will all take the upper branch and X and Z both look like candidates for the same position in C . At a given time, the nodes reside at the positions shown in Table I. Y finishes processing in B , so it can proceed to C . All it needs to do is register itself with the (currently empty) `SynchroCell` and suspend until the `SynchroCell` decides which nodes it needs. Before suspending, Y now increments its index to $[2, \mathcal{P}_1, 2]$, but because $[2, \mathcal{P}_1, 1] > [1]$, it does not need to inform Z of this. X and Z progress simultaneously to B , which means that they both change to index $[2, \mathcal{P}_1, 1]$. X must now message Y that the new minimum is $[2]$.

Upon receiving a message, Y comes out of suspension and checks with the `SynchroCell`. The fact that $Y.\text{min} = [2] < [2, \mathcal{P}_1, 2] = Y.\text{index}$ means that there may be predecessors of Y that can arrive at the `SynchroCell` in the future. Y should, however, still message Z that the latter’s predecessor minimum is now $[2]$, before going back to suspension.

Z finishes processing at B and goes on to C , where it registers with the `SynchroCell`. The `SynchroCell` sees Z as a candidate, so Z , like Y before it, is suspended.

When X hits the `SynchroCell` next, it will see that Z is registered as a candidate for the same position. Two things can now be concluded from the propagated minimum values. Firstly, because $Z.\text{min} = [2] < [2, \mathcal{P}_2, 1] = X.\text{min}$, X is a predecessor of Z , so Z can be released and X should be kept as candidate. Secondly, no predecessor of X can ever reach the `SynchroCell`, because $[2, \mathcal{P}_2, 1] \not\prec [2, \mathcal{P}_1, 1]$, so the record stored on node X can be consumed and the node replaced by a simple indirection node.

Before the handler thread of X dies, it propagates the minimum value registered at X to Y . This, again, takes Y out of suspension. Upon checking itself with the `SynchroCell`, it can be concluded for Y that no predecessor can ever reach the `SynchroCell`, in the same way that it was concluded for X . The `SynchroCell` merges the record it consumed into the record on node Y and releases Y , which now flows out of the network.

IV. RELATED WORK

To the best of our knowledge, the problem presented here is rather unique among the synchronous and asynchronous stream processing networks and languages [5], [6], [7]. However, the problem of re-establishing the order of records does arise in other contexts, such as packet sequencing problems in data networks [8].

In data networks, packets transmitted in a particular sequence could reach the destination via different routes due to policies enforced by the network elements, for example, route selection based on network congestion. As a result, packets may arrive at the destination out-of-order and must be reordered before being passed to the presentation layer. Simple numbering schemes, which number the packets in transmission order, work very well and this is often used in conjunction with retransmission requests. Towards this, data networks traditionally rely on queue-based solutions for re-establishing the order of packets.

In contrast, in `S-NET`, re-establishing the order of records in a stream does not occur only at the output of the record. Order is required at certain points *within* the network (at stateful components) and not all records will flow through that point, i.e. the order of only a subset of all the records in the stream must be re-established. This local, partial reordering problem distinguishes itself even further because of the multiplicity property (which does not exist in data networks). These requirements are not met by simple TCP-like numbering schemes.

`StreamIT` [7] is a stream processing language/network where ‘filters’ perform the functionalities of boxes in `S-NET`. All filters are stateless and each filter is equipped with a queue. The only occasion where reordering matters within a `StreamIT` network is directly after a stream split. In such cases, the network performs deterministic merging/ordering of the stream. In `StreamIT`, the stream split (parallel composition) is simplified by a *splitter-joiner* pair and all parallel

branches carry out the same operation. A splitter-joiner pair is statically configured to use a pre-selected stream distribution and merging scheme. As a result, the temporal order and semantic order are only decoupled locally between splitters and joiners, eliminating any need for a reordering mechanism beyond joiners.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a stream-order relaxed execution model for asynchronous stream languages. In proposing the model, we have made the following contributions:

- We have proposed a simple yet elegant scheme for maintaining the semantic order of the stream—a linked list. The choice leaves us with added advantages such as maintenance-free semantic order enforcement, the ability to support multiplicity and a simple thread communication scheme.
- We have derived a powerful network enumeration, which enables us to deduce the reachability of records between different components of a network.
- We have provided an algorithm that guarantees the correct execution of networks with stateful components, using only the network enumeration and local inter-thread communication.

These contributions underpin our execution model, which provides improved concurrency by decoupling the semantic and temporal orders of records. The DFT execution model can be improved in the following ways (future work):

- The initial semantic order is preserved under *all* circumstances. This means that all mergers are deterministic. Originally, non-deterministic variants of network combinators were used to locally relax the semantic order, whereby exposing some concurrency. Since the DFT execution model exposes a superset of that concurrency, non-determinism no longer helps to increase it. Future work will show that non-determinism *can* be a relaxation of resource usage (esp. memory).
- How to schedule the threads of the DFT model deserves special attention. The daunting amount of threads spawned in high-throughput networks require specialized scheduling techniques to prevent resource contention.
- Although confidence is high that the DFT algorithms are deadlock free (inter-thread communication is unidirectional and SynchroCells are the only shared resource—at most one per thread), a proof—or at least a more detailed analysis—would add significantly to the appeal of the DFT execution model.

ACKNOWLEDGEMENTS

The development of S-NET is funded by the European Union through the Framework VI Integrated Project *ÆTHER*³, *Self-adaptive Embedded Technologies for Pervasive Computing Architectures*.

³<http://www.aether-ist.org/>

REFERENCES

- [1] C. Grelck, S.-B. Scholz, and A. Shafarenko, “A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components,” *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.
- [2] C. Grelck and F. Penczek, “On Implementing S-Net,” in *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages (IFL’07), Freiburg, Germany*, ser. Technical Report 12-07, O. Chitil, Ed. University of Kent, Computing Laboratory, Canterbury, England, UK, 2007, pp. 531–533.
- [3] A. Shafarenko, C. Grelck, and S.-B. Scholz, “Semantics and type theory of S-Net,” in *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL’06), Budapest, Hungary*, ser. Technical Report 2006-S01, Z. Horváth and V. Zsók, Eds. Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, 2006, pp. 146–166.
- [4] H. Cai, S. Eisenbach, A. Shafarenko, and C. Grelck, “Extending the S-Net Type System,” in *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS’07), Paris, France*, 2007.
- [5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data-flow programming language LUSTRE,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.
- [6] G. Berry and G. Gonthier, “The ESTEREL synchronous programming language: design, semantics, implementation,” *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, Nov. 1992.
- [7] W. Thies, M. Karczmarek, and S. P. Amarasinghe, “StreamIT: A language for streaming applications,” in *Proceedings of the 11th International Conference on Compiler Construction (CC), Grenoble, France, April 8-12, 2002*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 179–196.
- [8] S. Moore and C. Siller, “Packet Sequencing: A Deterministic Protocol for QoS in IP Networks,” *IEEE Communications Magazine*, vol. 41, no. 10, pp. 98–107, Oct. 2003.