# PS-NET - A Predictable Typed Coordination Language for Stream Processing in Resource-Constrained Environments

Raimund Kirner, Sven-Bodo Scholz, Frank Penczek, Alex Shafarenko
Department of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
{r.kirner, s.scholz, f.penczek, a.shafarenko}@herts.ac.uk

***Abstract*— Stream processing is a well-suited application pattern for embedded computing. This holds true even more so when it comes to multi-core systems where concurrency plays an important role. With the latest trend towards more dynamic and heterogeneous systems there seems to be a shift from purely synchronous systems towards more asynchronous ones. The downside of this shift is an increase in programming complexity due to the more subtile concurrency issues. Several special purpose streaming languages have been proposed to help the programmer in coping with these concurrency issues. In this paper, we take a different approach. Rather than proposing a full-blown programming language, we propose a coordination language named PS-Net. Its purpose is to coordinate existing resource-bound building blocks by means of asynchronous streaming. Within this paper we introduce code annotations and synchronisation patterns that result in a flexible but still resource-boundable coordination language At the example of a raytracing application we demonstrate the applicability of PS-Net for expressing the coordination of rather dynamic computations in a resource-bound way.**

***Keywords*-stream processing; embedded systems; multi-core; resource-constrained;**

## I. Introduction

Stream processing is an apt metaphor of embedded computing. Indeed, owing to the generally static nature of streams connecting processing nodes, a higher degree of predictability may be achieved in representing embedded systems as stream-processing networks than with the dynamism of imperative and object-orientated milieux, where control and data can be passed from any point in the program to a given program unit provided that it is visible in that point's name space. Traditionally, stream processing is understood through the prism of the single-instruction multiple-data (SIMD) perspective. The paradigm itself is seen as a version of the latter with a different connectivity principle (streams instead of shared memory). This understanding is upheld by a number of projects, notably Stanford-based Merrimac [1] and Brook [2]. As an extreme form of this approach, one should mention strictly time-controlled synchronous solutions such as Giotto [3], [4], [5] and Scade [6], [7]. Here, the trade-off between predictability and efficiency is tilted towards predictability.

Generally, stream processing need not to be SIMD or even synchronous. In the most abstract sense, it is a representation of a program in terms of a static network of entities, each completely encapsulated and interacting with the rest only via its input and output streams. When streaming is to be used as a construction principle for larger systems, an asynchronous approach would usually be favoured, i.e., apriori unknown production rates and message arrival times. An example of this can be found in the language StreamIt [8], which has asynchronous messages and bounded nondeterminism. The most recent offering of an asynchronous streaming language comes from the project WaveScript [9] whose aim is essentially to integrate the network view and the local, synchronous view within one language with streams as first-class entities. Since this is a general-purpose streaming language, here, too, application programming concerns (i.e., algorithm correctness, ease of software evolution and accommodation of a continually changing specification) are intertwined with a whole spectrum of distributed computing concerns, such as work division, synchronisation, and load balancing, within a single level of program representation.

In our view, a more productive approach to applying the stream processing paradigm to embedded computing is to keep the concerns separated, with predominantly computational parts of the application represented as black boxes being written in a conventional programming language and with stream communication, data synchronisation and concurrency concerns being taken care of by a coordination language. We specifically focus on S-Net [10], [11], where we believe the above programme has been realised to the fullest possible extent.

The ground level of S-Net comprises stream-processing nodes represented as C-functions (or functions written in an array processing language, such as SAC [12]). These computational entities are called "boxes" and they communicate with the S-Net world via a single input and a single output stream. Data elements on these streams are represented as non-recursive record structures.

In a way, the set of boxes for a given application represents the nodes of a specialised virtual machine. The coordination program can abstract from the box functionality, the more so that the records streamed between boxes are being completely encapsulated as well: all the coordination level can see is field labels and some auxiliary integer-valued tags. This opens up an avenue towards sensible software engineering of embedded systems, where subject experts could be engaged in writing box code and describing the computational process informally in terms of record structures and box connections, and where *concurrency engineers* could be in a position to write, debug and optimise the coordination code with the experts' minimum assistance. That is the most attractive feature of the coordination approach compared with the competing strategies cited earlier.

However, this is not without some new problems either. The fundamental assumption of S-Net is that the application is not resource bounded. While not unreasonable in a large-scale distributed computing domain, this assumption is completely unrealistic in most embedded systems, where, if coordination has a chance, it must be essentially resource driven. This means that the placement of boxes on the system must be governed by the availability of cores and a predictive estimate of their load, which in turn means that the coordination layer must be in possession of accurate information about how much processing and communication is required for the completion of each task. By contrast, S-Net achieves its separation of concerns by relying on asynchronous dynamic adaptation: nondeterministic stream mergers, for instance, are assumed to merge in the order of record arrival, thus economising buffering space and reducing latency. Worse still, more dynamic features of S-Net namely its serial and parallel replication facilities, are not even a priori bounded since the boxes are not assumed to have the knowledge of, or the ability to communicate, the overall application design.

We set ourselves the challenge of finding a way to reconcile the need for dynamic behaviour and with the necessity to project tight enough bounds on the platform as far as the resource requirements. The S-Net facilities must therefore be curtailed to allow for static specification of various computational bounds, such as the maximum unfolding of the replicators, the maximum production rate of the boxes, the maximum correlation between the output rates dependent on a single input stream, etc.

In this paper we examine the relevant coordination facilities of S-Net in Section II and work out in Section III what needs to be modified and how so that S-Net may become usable with embedded applications. The result is a new language, called PS-Net, which is described in Section IV. Section V shows an example of how to write resource-bounded programs in PS-Net. Section VI concludes the paper.

## II. Stream-Processing with S-Net

In order to present a specialised variant of S-Net that is resource-boundable, we first give a very brief overview of the language. A detailed description of S-Net can be found in the literature [10], [11].

The central philosophy of S-Net is to separate the coordination of concurrent data streams from the computational part. Computations on data are not expressible in S-Net as such, but are written in a conventional programming language. These pieces of "foreign" code are embedded into *boxes* and are given an extremely simple API to communicate with the surrounding S-Net. The API allows them to receive data from a single input stream via the normal parameter-passing mechanism, and which provides a small number of library functions for outputting data down the single output stream, both streams being anonymous. Boxes may not have a persistent internal state and consequently can only process input data individually. Nor can they access each other's state in any way during the processing: there are no global variables or inter-box references. Instead, the output records are streamed by the coordination layer of S-Net according to a coordination program that defines the streaming topology, how the streams are split and merged, and how individual records are split, merged and routed to their intended destinations.

In order to guarantee interoperability of computational entities and all different parts of a streaming network, the coordination of data flow in S-Net is analysed by means of a type system and inference mechanism. The type system of S-Net is based on non-recursive variant records with *record subtyping*. Each record variant is a possibly empty set of named record entries, where a record entry is either a field or a tag. The values of fields are only accessible by the box implementations, while the tags are integer variables whose values can also be accessed and manipulated by both, the S-Net program and the box implementations. To separate tags from fields, the tag names are surrounded by angular brackets, e.g., `<a>`. Tags allow to use some logic operations to control the flow of data. The following is a variant record type that encompasses both rectangles and circles enhanced with a tag `<id>`: `{x,y,dx,dy,<id>} | {x,y,radius,<id>}`. Each S-Net network or subnetwork has a type signature, which is a non-empty set of variant record type mappings each relating an input type to an output type. For example, a network that maps a record `{a,b}` to either a record `{c}` or a record `{d}` or maps a record `{a}` to a record `{b}` has the following type signature: `{a,b}->{c}|{d}, {a}->{b}`. S-Net also supports subtyping. For example, `{a,b}` is a subtype of `{a}`.

As with conventional subtyping, in S-Net a network or box also accepts input data being a subtype of the network's or box's input type. Those record entries of the subtype that do not match a record of the box's input type simply bypass the network or the box and are joined with the produced output. Thus, an S-Net box with the type signature `{a}->{b}`, for example, also accepts input data of the type `{a,c}`, like a type signature `{a,c}->{b,c}` but where the record field `c` simply bypasses the box. This feature is called *flow inheritance*.

### A. Stream-manipulation with Filter Boxes

In S-Net, so-called *filter boxes* are used to perform manipulations of the data stream, like elimination or copying of fields and tags, adding tags, splitting records, and simple operations on the tag values. Filter boxes are expressed in square brackets and consist of a semicolon-separated list of filter actions on the right side of the transformation arrow. For example, the filter box `[{a,b,c} -> {a};{b,<t=1>};{b=c,<t=2>}]` takes records of type `{a,b,c}` and splits them into three output records: one with the field `a`, one with the field `b` extended by a tag `<t>` with the value 1, and one with the field `c` renamed to `b` and extended by a tag `<t>` with the value 2. Though the last two output messages contain the same field name `b`, they can still be processed differently at S-Net level due to their different value of tag `<t>`.

### B. Network Combinators in S-Net

S-Net consists of the following four combinators to combine networks or boxes. For the description of them we assume that we have two networks $net_1$ and $net_2$ that we want to combine.

1) **Serial Composition** ($net_1$ `..` $net_2$)**:** This allows to combine two S-Net networks or boxes in a sequential fashion. Though sequential in its dataflow, in the context of stream-processing this provides parallel

processing in the form of pipelined execution. The code $net_1$ `..` $net_2$ essentially forms a pipeline with the stages $net_1$ and $net_2$.

2) **Parallel Composition** ($net_1$ `|` $net_2$)**:** This allows to combine two S-Net networks or boxes in a parallel fashion, providing concurrent execution. The code $net_1$ `|` $net_2$ describes a split of data flow between the routes of networks $net_1$ and $net_2$. If $net_1$ and $net_2$ have different type signatures then the type system of S-Net will route the data to the best-matching input type, otherwise the choice is non-deterministically.
3) **Serial Replication** ($net_1$ `*` {`out`})**:** The serial replication (subsequently also called *star operator*) creates a pipeline dynamically by replicating the given network along a series composition till the output is a (sub)type of the exit pattern, where in this case the output is forwarded as the output of the replication operator. $net_1$ `*` {`out`} means that the data flow through a series composion of replicas of the network $net_1$ till the type of the output is a (sub)type of {`out`}.
4) **Parallel Replication** ($net_1$ `!` {`<id>`})**:** The parallel replication is the dynamic variant of parallel execution, where a given network is replicated dynamically controlled by the value of a tag in the data records. $net_1$ `!` {`<id>`} means that for each different value of the tag `<id>` of the incoming data records an exclusive path through a replica of the network $net_1$ is dynamically created.

Note that the combinators `|`, `*`, `!` have an out-of-order semantics on data routing, while `||`, `**`, `!!` are their order-preserving variants.

### C. Synchronisation with Synchro-cells

Above S-Net operations are all asynchronous and stateless operations, allowing for an efficient concurrent processing of data streams. To synchronise the arrival of different message types, the so-called *synchro-cell* is used, which is the only stateful box in S-Net. The synchro-cell is the only means in S-Net to combine two records into a single record. The synchro-cell consists of an at least two-element comma-separated list of type patterns enclosed in `[|` and `|]` brackets. For example, the synchro-cell `[| {a}, {b,c} |]` composes two records `{a}` and `{b,c}` into a single output record `{a,b,c}`. As its state, a synchro-cell has storage for exactly one record of each pattern. When an arriving record finds its place free in the synchro-cell, it is stored in the synchro-cell, otherwise it is simply passed through. The synchro-cell is a one-shot operation, i.e., once all record patterns are filled, the composed output record is emitted and the synchro-cell from now on behaves like a simple connector passing all further messages through. To use synchro-cells in a continuous way on the input stream, it has to be nested within replication operators as described above.

## III. Discussion of Predictability

In the following, we discuss what features of S-Net are hard to bound for their resource consumption and we discuss how we address this problem in PS-Net to ensure boundability of resources. The following mechanisms of S-Net are hard to bound without doing an exhaustive whole-program analysis:

- The computational part of S-Net programs is implemented in boxes. Regarding the boundability of the dynamic resource allocations, it is, of course, necessary, that the box implementations are simple enough to bound their resource requirements. However, the box implementation is outside the scope of our design of the resource-boundable coordination language PS-Net.
- In S-Net a network or box may write an arbitrary number of output messages as the type signature does not restrict them. Thus it is not know how much system load can be created within the network. This makes it hard to bound extra-functional properties such as execution time. Our solution for PS-Net is to extend the type signature with the multiplicity of the different output messages.
- The parallel composition in S-Net (`|`,`||`) features a non-deterministic choice, whose behaviour cannot be analysed precisely at language level, which makes it challenging to bound extra-functional properties such as execution time.
- The number of parallel replications in S-Net (`!`,`!!`) depends on the possible values of the replication-controlling tag value, which is hard to bound in general. In order to bound the number of dynamically created replicas for the parallel replication operator, we have to know the possible value range of the index tag. For PS-Net we extend the parallel replication combinator with an annotation about the maximum range of the index tag.
- The number of serial replications in S-Net (`*`,`**`) depends on the dynamic creation of the exit type, which is hard to bound in general. In order to bound the number of dynamically created replicas for the serial replication operator, we have to know when latest the exit pattern is produced. For PS-Net we extend the serial replication operator with an annotation of the maximal number of created replications.

### A. Synchronisation Mechanisms

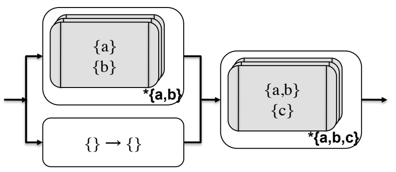
The synchronisation issues deserve a special discussion. On the one side the synchro-cell of S-Net has a single-shot semantics which is no problem at all to account for its maximum resource usage. However, as already said, the synchro-cell is typically embedded into a serial replication with infinite replications. This infinite replication is not a problem in S-Net, since every replica with a synchro-cell that has already shot is automatically discarded and automatically replaced by a direct stream connection.

However, our general solution of making the serial replication boundable by adding an annotation about the maximal number of created replications, is unfortunately not compatible with the use of the S-Net synchro-cell, as this would rely on an infinite replication count.

Our solution for PS-Net is to avoid the combination of synchro-cell and serial replication and instead use special synchronisation constructs for use patterns of synchro-cells. We have actually identified two major use patterns for synchro-cells. They stem from the need to either synchronise a statically fixed number of records or to synchronise a dynamically varying number of records, respectively.

In the former case, the records that are to be combined can be encoded by different types. This facilitates an implementation of the synchronisation as a cascade of synchro-cells embedded in serial replications.

Figure 1 shows an example for such a synchronisation.

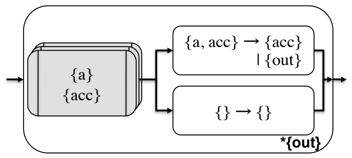


```
([|{a},{b}|]*{a,b} | []) .. [|{a,b},{c}|]*{a,b,c}
```

Fig. 1. Synchronising records of type {a}, {b}, {c}.

There, three records are being synchronised each, one record of type {a}, one of type {b}, one of type {c}. The first synchro-cell within a star combines records {a} and {b}. Any records that are neither {a} nor {b} are bypassed by means of the identity filter which is parallel to the synchroniser for {a} and {b}. Subsequently, the bypassed records of type c are synchronised with the combined records of type {a, b}. Again, this second synchro-cell is directly embedded into a star to enable repeated synchronisation.

In the second case, i.e., when we deal with a statically not determined number of records to be synchronised, a type encoding of the individual components of the record to be combined is no longer possible. Instead, a stepwise synchronisation needs to be applied to a substream of records of the same type. The emerging result record needs to be propagated from one synchronisation to the next similar to an accumulator within a folding operation. When to terminate such a folding process needs to be determined either by the folding operation itself or by the use of ”separation records“ of different type in the stream. Figure 2 shows an example for such a multi-synchronisation. Here, the folding box itself



```
net multi_sync {
  box fold( (a, acc) -> (acc) | (out) );
} connect ( [|{a},{acc}|] .. ( fold | [] ))*{out};
```

Fig. 2. Synchronising multiple records of type {a}.

determines when to emit a value by producing a record with a field out rather than acc. This network furthermore assumes that the initial value for each synchronisation comes in as a record containing acc. Note, that the empty filter that is parallel to the fold box serves as a by-pass for subsequent records of type {a} or type {b} so that they can be fed into subsequent unfoldings of the star combinator.

Within a range of S-Net applications [13], [14], [15], we could observe various different formes of synchro-cell uses within serial replication. However, it turns out that all of them adhere to one of the two use cases above and can be expressed by nestings of these two pattern. Therefore we capture those two pattern as two new building blocks in PS-Net, named syncq and fold, respectively.

## IV. Resource-Bounded PS-Net

In this section we introduce new language constructs that are boundable. Further we introduce annotations for existing S-Net language constructs to make them boundable. These annotations might be written by the programmer or being automatically derived by program analysis. All the annotations have the form <| *AnnotExpr* |> where *AnnotExpr* can be of the following forms:

| | |
|---|---|
| *Num* | ...specifies a constant value |
| *Num* : | ...specifies a lower bound |
| : *Num* | ...specifies an upper bound |
| *Num* : *Num* | ...specifies an interval |

### A. Multiplicity of Box Messages

For PS-Net we extend the type signature with an annotation about the multiplicity of messages. For example, the following box signature declaration `box foo ((a,b) -> (c)<|2|> | (d)<|1:3|>);` specifies that for each processing of on input record {a,b} the box creates exactly two output records of type {c} and between one and three output records of type {d}. Note that the records of box signatures are written in round brackets to distinguish them from network type signatures, since for the box signature the order of record entries matters.

### B. Bounded Parallel Replication

The range of the index tag determines the number of different dynamically created parallel replicas. Assuming that an index range will always start from zero, we extend the parallel replication with an annotation of the upper bound of replications $k$, resulting in an index range from $0 \dots k-1$. replication index range. For example, to specify that a network can be at most replicated four times (i.e., index range 0 to 3), we can write:

```
network ! <tag><|4|>;
```

Note that the total number of replications can be higher if the network is nested within another network that is replicated as well.

### C. Bounded Serial Replication

We extend the serial replication operator with an annotation of the maximal number of created replications. For example, to specify that a network can be at most replicated three times (i.e., a pipeline of length three), we can write:

```
network * {out}<|:3|>;
```

### D. Synchronisation with the syncq operator

The first use case of synchronisation (Figure 1) can be abstracted by means of a *synchro-queue*, which repeatedly synchronises records of two flavors defined by means of two type pattern.

Provided that the synchronic distance [16] between the two flavors is bounded, such an operator can be implemented as a finite queue whose length does not exceed that bound. We introduce synchro-queues as a new operator

syncq[| $p_1$, $p_2$ |]<| $sd$ |>

where $p_1$ and $p_2$ denote type pattern to be synchronised, and $sd$ denotes an upper bound for the synchronic distance between the pattern $p_1$ and $p_2$ on the input of this operator. For example, if we want to synchronise records of type {a} and {b} knowing that the synchronic distance between them is at maximum 4, then we can write:

`syncq[| {a}, {b} |]<|4|>`

Formally, the semantics of the `syncq` operator is defined by the following equivalence:

$$\texttt{syncq[|}\ p_1\texttt{,}\ p_2\ \texttt{|]<|}\ sd\ \texttt{|>} \equiv$$
$$\texttt{[|}\ p_1\texttt{,}\ p_2\ \texttt{|]*\{}p_1\texttt{,}\ p_2\texttt{\}}$$

Note here, that the star version on the right hand side of the equivalence potentially requires unbounded resources. Only the annotated synchronic distance $sd$ ensures boundedness of the operator. Interestingly, a finite synchronic distance also implies that all the synchronised patterns have the same average arrival rate.

## *E. Synchronisation with the `fold` operator*

The second use case of synchronisation (Figure 2) can be abstracted into a generic *folding operation*. Here, we introduce a network combinator, which transforms a folding network with a signature ($a$, $acc$) -> ($acc$) into a network that subsequently synchronises a record of type $\{acc\}$ with an arbitrary number of records of type $\{a\}$, until a new record of type $\{acc\}$ arrives which triggers a new series of synchronisations.

Syntactically, we denote the fold combinator by

$$\texttt{fold[|}\ a\texttt{,}\ acc\texttt{,}\ N\texttt{,}\ Fold\texttt{|]}$$

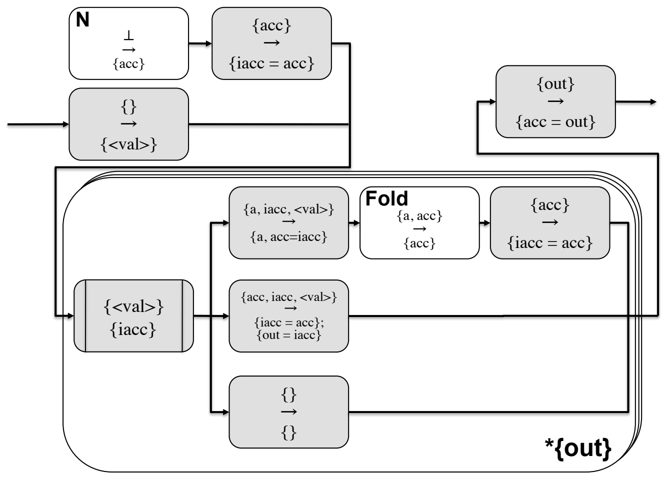
where $a$ and $acc$ denote the two different kinds of type pattern, $N$ denotes a network with a type signature $\perp$ -> ($acc$)<|1|> that provides the initial value for $acc$ and $Fold$ is a network of type ($a$, $acc$) -> ($acc$)<|1|> which implements the folding operation itself. For example, if we want to collect partial results of type {d} of a concurrent computation into result messages of type {res}, with `Init` being the network to create the initial {res} message and `Collect` being the network name of the fold operation that merges a partial result of type {d} with the current result message of type {res}, then we can write:

`fold[| {d}, {res}, Init, Collect |]`

The semantics of a network fold[| $a$, $acc$, $N$, $Fold$|] then is defined by the S-Net shown in Figure 3.

The main complexity of this network stems from the necessity to "restart" the folding process upon arrival of a new record of type {acc}. To achieve this, all incoming data is tagged with <val> upon arrival. In the core of the network, this data, i.e., either records of type {a, <val>} or of type {acc, <val>} are synchronised with the current state of the folding operator which is kept in an internal accumulator field `iacc`. Depending on the type of the synchronised record, either the `Fold` network is applied and the internal accumulator is updated accordingly, or the current result is emitted via a record of type {out} and the internal accumulator is reset to the new value from the input field `acc`. Note here, that the overall fold combinator needs to be initialised with a record of type {acc} provided by the network `N`. Its value serves as initial state for the internal accumulator.

A key observation of this network is that for each incoming record the synchro-cell of the first incarnation of the star operator synchronises which transforms the entire inner network effectively into an identity function for the subsequent records. In combination with a multiplicity of 1 for the `Fold` network, this guarantees that the fold operator can be implemented in constant space.



```
( ( N .. [ {acc} -> {iacc = acc} ])
  || [{} -> {<val>}]
)
.. ( [| {<val>} , {iacc} |]
     .. ( ( [ {a, iacc, <val>} -> {a, acc=iacc} ]
              .. Fold .. [ {acc} -> {iacc=acc} ]
          )
          |  [ {acc, iacc, <val>} -> {iacc = acc};
                                     {out = iacc} ]
          |  []
        )
   ) * {out}
.. [ {out} -> {acc = out} ]
```

Fig. 3. Network implementing the fold operator

## V. EXAMPLE

We evaluate the presented approach by applying it to the well-known fork-join pattern that many image processing applications expose. An image is broken down into smaller chunks and an application specific processing algorithm is run on each chunk independently in an SIMD-like fashion. A merging stage collects all processed chunks, i.e. the sub-results, and reassembles a global result image.

Where previous experiments using S-Net in its standard form have shown that this class of applications lends itself nicely to the advocated programming model we are now in a position to reformulate existing code to guarantee resource-bounded execution in PS-Net. As a representative problem of this class we implemented a ray-tracing image processing application for which we have developed an implementation in standard S-Net with performance results that compete with hand-tuned C code [13].

The implementation of the original application is intended to run on general-purpose hardware and is specified as follows:

```
net raytracing {
  box splitter( (scene, <rr_upper>, <tasks>)
        -> (scene, chunk, <rr>, <tasks>, <fst>)
        |  (scene, chunk, <rr>, <tasks> ));
  box solver  ( (scene, chunk) -> (sub_res));
  net merger  ( (sub_res, <fst>) -> (pic),
                (sub_res)        -> (pic));
  box genImg  ( (pic)          -> ());
} connect splitter .. solver!<rr> .. merger
        .. genImg;
```

The splitter divides the scene into smaller sub-scenes (chunks) and tags all chunks with the number of overall produced sub-scenes. Each data element also carries an `<rr>` tag. This implements a round-robin scheduling using the ! combinator on the solver by tagging data elements with increasing integer values from $0, ..., $`<rr_upper>`$-1$ for `<rr>`. The first output is tagged with `<fst>` to initiate the merging process after the sub-scenes have been computed by the solver. The merging process is implemented as a sub-network of the following form:

```
net merger {
  box init  ( (sub_res, <fst>) -> (pic));
  box merge ( (sub_res, pic)   -> (pic));
} connect ( ( init .. [ {} -> {<cnt=1>} ] ) | [])
          .. ( [| {pic}, {sub_res} |]
               .. ( ( merge
                      .. [ {<cnt>} -> {<cnt+=1>}])
                  | []))*{<tasks> == <cnt>};
```

The init box is followed by a filter which adds a flag `<cnt>` initialised by the value 1. This flag is used to count the number of sub-scenes that have been incorporated into the result image already. Since only the first sub-scene needs to be processed by the init box, we also provide a bypass to the initialisation path for all the other records containing further sub-scenes.

After the initialisation, a star implements the merging with the remaining sub-scenes. In each unfolding (iteration) of the star the synchro-cell synchronises the accumulator held in {`pic`} with yet another sub-scene. The resulting joint record, containing the accumulated picture and a sub-scene to be inserted, is presented to the merge box which outputs the combined picture. The insertion of a new sub-scene is reflected in an increment of the flag `<cnt>` as defined by the subsequent filter. Once the counter equals the overall number of tasks, which is kept in another, flow-inherited flag `<tasks>`, the accumulated picture is output from the merger network.

In order to guarantee resource-boundedness of this implementation, we replace the parts of the application that make use of the general $\star$ and ! combinators by legal PS-Net constructs.

The splitting stage of the application is almost straightforwardly transformed. As we are not targeting general-purpose hardware, we use the `! <rr><|n|>` combinator and annotate the maximum number $n$ of computing resources the combinator is allowed to bind for solver instances. Because of the way we are implementing the merging process, which is detailed below, the splitter is not required to output the number of produces sub-scenes. Additionally, it also does not tag the first element. Instead, the splitter outputs the accumulator as first record for each decomposed scene.

With the PS-Net fold combinator we are able to re-implement the merging stage of the original application. The combinator's behaviour resembles the functionality of the merging stage when supplied with the merger box of the original application as fold-net argument. An initialiser network is not required, as we chose to have the splitter output all `pic` accumulators including the first one.

Putting it all together, the resource-bound version of the application is defined as follows (we chose 7 as an arbitrary resource limit for the ! combinator for illustration purposes):

```
net raytracing {
  box splitter( (scene, <rr_upper)) ->
                  (scene, chunk, <rr>) | (pic));
  box solver( (scene, chunk) -> (sub_res));
  box merger( (sub_res, pic) -> (pic));
  box genImg( (pic) -> ());
} connect splitter
       .. (solver!<rr><|7|> | [(pic) -> (pic)])
       .. fold[|{sub_res},{pic},_,merger|]
       .. genImg;
```

This network behaviour resembles that of the original implementation. The splitter outputs a variable number of sub-scenes and the solver is applied to these in parallel. The merging stage is wholly implemented by the fold combinator. But this implementation is guaranteed to be resource bound: The parallel replication is limited by an annotated upper bound. As the fold combinator is statically resource bound, we do not require multiplicity annotations on the splitter box.

This example has shown how the proposed coordination languages for stream processing can be used to model resource-constrained embedded applications. The stream-processing model itself has the benefit that it naturally combines the flexibility of asynchronous computation with a separation of concern between coordination and algorithmic programming.

## VI. Conclusion and Future Work

In this paper we have shown the development of the resource-boundable coordination language PS-Net for stream processing, starting from the S-Net language, which has been designed for the high-performance computing domain. On the one side we had to add annotations to certain language constructs, to make them resource-boundable. Such annotations might be written directly by the developer or may be derived automatically by program analysis. Further, we have introduced the *synchro-queue* and the *folding combinator* as resource-boundable synchronisation constructs. The resulting language allows to program dynamic stream-processing applications in a resource-bound way. As a future work we will implement PS-Net within the S-Net compiler, which is quite suitable for this implementation, since the new resource-boundable constructs introduced for PS-Net can be implemented with S-Net constructs. These S-Net constructs would be non-resource-boundable in the general case, but become resource-boundable for the specific patterns derived from PS-Net constructs. Further, evaluations of resource consumption are planned to demonstrate the suitability of the PS-Net programming paradigm for embedded computing.

*Acknowledgments*

The research leading to these results has received funding from the IST FP-7 research project ”Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)”.

## References

[1] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labont, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, “Merrimac: Supercomputing with streams,” in *Proc. ACM/IEEE Conference on High Performance Networking and Computing (SC’03)*, Phoenix, Arizona, USA, Nov. 2003.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: Stream computing on graphics hardware,” in *Proc. ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*, Los Angeles, USA, 2004, pp. 777–786.

[3] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: A time-triggered language for embedded programming,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

[4] T. A. Henzinger, C. M. Kirsch, and S. Matic, "Composable code generation for distributed Giotto," in *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2005.

[5] A. Ghosal, D. Iercan, C. M. Kirsch, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli, "Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code," in *Online Proc. Workshop on Automatic Program Generation for Embedded Systems (APGES)*, 2007.

[6] F.-X. Dormoy, "Scade 6: A model based solution for safety critical software development," in *Proc. 4th International Conference on Embedded Real Time Software (ERTS)*, Toulouse, France, 2008.

[7] E. Technologies, "SCADE suite," web page (http://www.esterel-technologies.com/products/scade-suite/), accessed in Jul. 2010.

[8] B. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. 11th International Conference on Compiler Construction (CC'02)*. London, UK: Springer Verlag, 2002, pp. 179–196.

[9] R. Newton, L. Girod, M. C. abd Sam Madden, and G. Morrisett, "WaveScript: A case-study in applying a distributed stream-processing language," Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, Cambridge, USA, Technical Report MIT-CSAIL-TR-2008-005, Jan. 2008.

[10] 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.

[11] A. Shafarenko, S.-B. Scholz, and C. Grelck, "Streaming networks for coordinating data-parallel programs," in *Perspectives of System Informatics, 6th International Andrei Ershov Memorial Conference (PSI'06), Novosibirsk, Russia*, ser. Lecture Notes in Computer Science, I. Virbitskaite and A. Voronkov, Eds., vol. 4378. Springer Verlag, 2007, pp. 441–445.

[12] C. Grelck and S.-B. Scholz, "SAC: A functional array language for efficient multithreaded execution," *International Journal of Parallel Programming*, vol. 34, no. 4, pp. 383–427, 2006.

[13] F. Penczek, S. Herhut, S. Scholz, A. Shafarenko, J. Yang, C. Chen, N. Bagherzadeh, and C. Grelck, "Message driven programming with s-net: Methodology and performance," in *3rd International Workshop on Programming Models and Systems Software for High-End Computing (P2S2'10), San Diego, USA*, 2010, to appear.

[14] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrere, and E. Lenormand, "Parallel signal processing with s-net," *Procedia Computer Science*, vol. 1, no. 1, pp. 2079 – 2088, 2010, iCCS 2010. [Online]. Available: http://www.sciencedirect.com/science/article/B9865-506HM1Y-88/2/87fcf1cee7899f0eeaadc90bd0d56cd3

[15] C. Grelck, J. Julku, and F. Penczek, "Distributed S-Net," in *Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA*, M. Morazan, Ed. Seton Hall University, 2009.

[16] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr 1989.