# S-Net: A Typed Stream Processing Language — Draft —

Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko

University of Hertfordshire
Department of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{c.grelck,s.scholz,a.shafarenko}@herts.ac.uk

**Abstract.** We propose a view on a data-processing application as a typed streaming network. The arcs of the network represent record-valued data streams and the nodes encapsulate recurrence relations on them. We propose a type system in which both the arcs and the nodes are statically subtyped, with the overall subtyping consistency of the network assured by type reconciliation algorithms. The proposed type system makes extensive use of a homomorphically-restricted subtyping, which, on the one hand, provides for generic node specification and, on the other, supports efficient type inference and type reconciliation.

## 1 Introduction

Component technology is crucial to implementing large systems on chip (SoCs). Indeed the complexity of on-chip solutions can only be overcome by decomposition and abstraction. The former requires application-specific building blocks, the latter demands that formal interfaces be set up between the blocks and the connecting infrastructure. This, of course, is not unique to systems on chip; any object-orientated technology would be based on similar ideas. What is typical of the SoCs is the static nature of the connection between the blocks and the increased role of provable properties in assuring the required functionality of the whole system. One way of viewing a static component network is associated with the concept of stream processing.

This paper proposes a component technology for stream processing, which we have named S-Net. An S-Net is a recursively nested single-input, single-output (SISO) streaming network that combines SISO processing boxes and coordinates their interaction. Processing boxes are atomic: they do not expose any internal content and, hence, are completely "black". These atomic boxes are entirely stateless and strictly operate in a input-process-output work cycle. Upon receiving a data item on its input stream an atomic box produces none, one or more data items on its output stream. The functional properties of atomic boxes enable them to be be deployed cheaply and moved and replicated at will, without giving rise to data integrity concerns.

S-Net is in fact a coordination language: It provides means to describe the orderly behaviour among boxes and the streaming network used for communication between boxes. Atomic boxes are implemented externally using an appropriate *box language*. Functional languages are particularly suitable for this purpose as they inherently adhere to the restrictions imposed by the interface. Nevertheless, imperative box languages may be used as well, but require some discipline by the programmer.

Atomic boxes communicate with each other and with the execution environment solely by means of data received and sent via typed streams. S-Net allows atomic boxes to be composed into SISO networks, which recursively form composite boxes in further network layers. Composition of boxes involves splitting and merging communication streams depending on their types. It is described using *network combinators*, that are inspired by Stefanescu's network algebra [1].

The restriction to a single input and a single output stream allows us to use variant types to capture data provenance under a type system. This makes the network topology a type issue alongside all the standard type issues, and opens up an avenue for comprehensive subtyping, which is what usually makes a component technology so effective in the object-oriented world. Still, the multiplicity of input and output streams as often found in stream processing can easily be mimicked: one may think of an SISO entity as one with all the input streams interleaved into a single stream and all the output streams similarly de-interleaved.

S-Net networks are asynchronous: an entity's output is assumed to be buffered. When processing is done by several components whose results must be combined, a synchronisation facility is generally required. It is introduced in the form of a SISO synchrocell, which is the only kind of "stateful" box in an S-Net. A syncrocell expects records of several types to appear at its input; it combines them into a joint record and outputs the result. The internal state of a synchro-cell is made up by the records waiting to be synchronised. Note that synchro-cells, though "stateful", have no computation to perform, whereas atomic boxes have no state, but can compute.

Finally, we propose genericity and specialisation mechanisms on the basis of static record subtyping. These mechanisms make it possible to statically optimise streaming networks with generic components. They also enable the component designer to provide several versions of a box depending on a subtype. Crucially, S-Net does not require explicit subtype declarations; a subtype inference algorithm is applied to determine the most appropriate subtype.

Data items sent via streams are organised as non-recursive, tagged variant records with arbitrary non-record fields. Consequently, the types associated with streams in an S-Net network are non-recursive, tagged variant record types. Elementary types are effectively opaque to S-Net. Since all actual data is produced and consumed by box language programs, only the box language code knows about how to interprete the data. As far as S-Net is concerned atomic record fields and the corresponding data travelling along the streams are as opaque as the atomic boxes themselves.

Tagged variant records allow a single record to alternatively store different fields. For instance, a geometric body type can include a sphere record with the fields *centre* and *radius*, as well as an ellipsoid record with the *centre* and three axes, and a cone with a *centre*, *hight* and an *apex* angle. A box may be capable of processing all these shapes, in which case a union type is required. To distinguish different members of the union type, which we shall call *variants* hereinafter, S-Net uses pattern-matching and tags.

Subtyping is an important adaptation mechanism of a component technology. An S-Net box may be capable of processing more variants than there are in the incoming typed stream, which should not cause trouble. Likewise, if a box receives a valid variant extended with additional fields, these fields should simply be ignored and passed on to the output so that a further box may process them. These commonsense considerations provide the motivation for subtyping. The first two of them constitute conventional record subtyping, whereas the third one is, to our knowledge, a new concept, which we call *flow inheritance*. It is fundamental to S-Net and is used extensively.

The remainder of this paper is organised as follows. Section 2 introduces record types and record subtyping as fundamental concept of S-Net. Then, we introduce S-Net as a programming language in Section 3. Section 4 provides an illustrative example before we sketch out related work in Section 5 and conclude in Section 6.

## 2 Record types and record subtyping

### 2.1 Record types

The type system of S-Net supports nonrecursive variant records with *record subtyping*. As formally defined in Fig. 1, a *record type* in S-Net is a possibly empty set of anonymous *variants*. Each variant again is a possibly empty set of named *record fields*. We distinguish two different kinds of record fields: *value fields* and *tags*. A value field is characterised by its *field name* and, as the name suggests, is associated with some value at runtime. This value, however, is opaque to S-Net and may only be generated, inspected or manipulated by using an appropriate box language. Likewise, a tag carries a name, but is associated with an integer value, that is visible to both: the box language code and S-Net.

| | | |
|---|---|---|
| *Type* | ⇒ | *TypeName* \| { [ *Variant* [ , *Variant* ]* ] } |
| *Variant* | ⇒ | *VariantName* \| { [ *Field* [ , *Field* ]* ] } |
| *Field* | ⇒ | *FieldName* \| < *TagName* > |
| *TypeDef* | ⇒ | **type** *TypeName* := *Type* ; |
| *VariantDef* | ⇒ | **variant** *VariantName* := *Variant* ; |

**Fig. 1.** Syntax definition of S-Net types and type definitions

S-Net supports non-recursive abstractions on types. Using the key word `type` an identifier may be bound to a type specification. As an example for a record type

```
type body := { {<Triangle>, col, x1, y1, dx2, dy2, dx3, dy3},
               {<rectangle>, col, x1, y1, dx2, dy2},
               {<circle>, col, x1, y1, r} };
```

defines a type `body` for the representation of geometric bodies, which are either triangles, rectangles or circles. Each body has a color (`col`), coordinates of a reference point (`x1` and `y1`) and varying numbers of further coordinates, e.g. edge lengths of a rectangle (`dx2` and `dy2`) or the radius (`r`) of a circle.

Likewise, we may define abstractions on variants using the key word `variant`. For example the above type definition may equivalently be written as

```
variant triangle  := {<Triangle>, col, x1, y1, dx2, dy2, dx3, dy3};
variant rectangle := {<rectangle>, col, x1, y1, dx2, dy2};
variant circle    := {<circle>, col, x1, y1, r};
type body         := { triangle, rectangle, circle};
```

We distinguish two different kinds of tags: *binding tags* and *non-binding tags*. Whereas the names of the former start with a capital letter, the names of the latter start with a small letter. So, in the above example `Triangle` in fact is a binding tag while `rectangle` and `circle` are non-binding tags. Binding tags and non-binding tags behave differently with respect to record subtyping, as defined in the following section.

### 2.2 Record subtyping

As mentioned earlier, S-Net supports subtyping on record types. Record subtyping is based essentially on the subset relationship between collections of record fields.

**Definition 1 (record subtyping).**
*Record subtyping is defined by the following rules:*

1. *Let $BT(x)$ denote the set of binding tags in a variant $x$.*
2. *A variant $v_1$ is a subtype of a variant $v_2$, $v_1 \sqsubseteq v_2$, if*

$$v_1 \supseteq v_2 \wedge BT(v_1) = BT(v_2)\,.$$

3. *A record type $t_1$ is a subtype of a record type $t_2$, $t_1 \sqsubseteq t_2$, if*

$$(\forall v_1 \in t_1 \exists v_2 \in t_2) v_1 \sqsubseteq v_2\,.$$

With the above definition of record subtyping, the purpose of binding tags becomes apparent: They provide a means to explicitly control record subtyping. One record variant can only be a subtype of another if the two have the same set of binding tags. In contrast, non-binding tags with respect to record subtyping behave just like ordinary value fields. For instance, the variant

[0] The non-terminal symbols *TypeName*, *FieldName* and *TagName* uniformly refer to identifiers. We only distinguish them here for illustration.

```
{<Triangle>, <colored>, x1, y1, dx2, dy2, dx3, dy3, color}
```

is a subtype of the variant tagged by `<Triangle>` of the definition of type `body` above. It contains exactly the same binding tags (`<Triangle>`) and all other record fields of the previous variant plus an additional non-binding tag `colored`. Note that the sequence of fields in the definition is irrelevant as variants are effectively sets of fields. Here is a subtype of the type `body` defined above:

```
type body1 := { {<circle>, color, x1, y1, r, shading},
                {<rectangle>, color, x1, y1, dx2, dy2, shading} }
```

Each variant is in subtype relationship to the corresponding variant of the supertype `body` identified by the same binding tag. To both variants we have added an additional value field `shading`.

With respect to record subtyping there are two special record types in S-Net: $\bot$ or $\{\}$, the record type with no variants, and $\top$ or $\{\{\}\}$, the record type with a single variant having no fields. Any type $t$ is a supertype of $\bot$: $(\forall t)\bot \sqsubseteq t$. Furthermore, any type $t$ free from binding tags is a subtype of $\top$: $(\forall t) BT(t) = \emptyset \implies t \sqsubseteq \top$.

### 2.3 Record type coercion

Now let us consider the issue of *record type coercion*. Coercion is achieved by throwing away the fields that are absent in the target type. This potentially causes an ambiguity whenever there is more than one suitable variant in the target type and, consequently, a choice of fields to dispose of. The above type `body` is a subtype of the type `anchored` defined as follows:

```
type anchored := { {<Triangle>, color},
                   {x1, y1},
                   {x1, y1, dx2, dy2} }
```

Indeed, each of the three variants of the (sub-)type `body` defined before can be shortened to one of the variants of (super-)type `anchored`. However, due to the special role of binding tags (here `<Triangle>`), the first variant of type `body` can only be coerced to the first variant of type `anchored`. In contrast, the `<rectangle>` variant of `body` can be coerced to either the second or the third variants of `anchored`. We resolve this ambiguity by defining coercion to always take the most specific candidate variant, i.e., if a variant $v$ may be coerced to variants $v_1$ or $v_2$ and $v_1 \sqsubseteq v_2$, then $v$ is effectively coerced to $v_1$. Hence, in the above example the `<rectangle>` variant of `body` is coerced to the third variant of `anchored` while `<circle>` is coerced to the second variant.

Still some of the ambiguity remains in that there can be two mutually incoercible targets for a given variant. For instance, in the type `confused`, defined as

```
type confused := { {x1,y1},
                   {dx2,dy2} }
```

there is a choice of variant to use for a `<rectangle>`. Only some targets for coercion can cause such ambiguities; the following definition introduces a uniqueness condition for type coercions:

| | | |
|---|---|---|
| *TypeSignature* | $\Rightarrow$ | { *Mapping* [ **,** *Mapping* ]* } |
| *Mapping* | $\Rightarrow$ | [ *Variant* ] **->** *Type* |

**Fig. 2.** Grammar for S-NET type signatures

**Definition 2 (complete record type).**
*A record type $\tau$ is called complete iff*

$$\forall v, w \in \tau : BT(v) = BT(w) \implies v \cup w \in \tau \, .$$

As in the definition of record subtyping above, $BT(x)$ denotes the set of binding tags of a variant $x$. For any pair of variants with the same set of binding tags a complete record type must have a third variant combining their fields. Note that all single-variant types are automatically complete.

### 2.4 Type signatures

Now, we are ready to define the concept of a *type signature*, i.e. the type associated with an S-NET box. Essentially, a type signature consists of an input record type and an output record type. The input record type $T_{in} = \bigcup_{i=1}^{n} \{v_i\}$ with variants $v_i$ specifies the records a box accepts for processing. The output record type $T_{out} = \bigcup_{i=1}^{n} \tau_i$ is in fact a collection of types $\tau_i$ with each $\tau_i$ being the type of records potentially created in response to receiving a record of variant $v_i$. In a conventional programming language would look very much like $T_{in} \rightarrow T_{out}$, but instead we use the following syntax to provide a subtyping structure:

$$\Sigma = v^{[1]} \rightarrow \tau^{[1]};\, v^{[2]} \rightarrow \tau^{[2]}; \ldots v^{[n]} \rightarrow \tau^{[n]} \, ,$$

where each $v^{[i]}$ is an input variant specification and each $\tau^{[i]}$ is a full type specifications of the output:

$$v^{[i]} = \{\phi_0^{[i]}, \ldots, \phi_{m_i}^{[i]}\};\;\; \tau^{[i]} = \{V_1^{[i]}, \ldots, V_{k_i}^{[i]}\};\; V_j^{[i]} = \{\Phi_{j,0}^{[i]}, \ldots, \Phi_{j,r_j}^{[i]}\} \, ,$$

with $\phi$ and $\Phi$ representing fields and $V$ variants. When a box $B$ is given an input stream $x$ of type $T$, every record of $x$ is coerced up from type $T$ to the input type $T_{in}$ of $B$ as described above. The concrete S-NET syntax for type specifications is given in Fig. 2.

Variant records in conventional languages have named variants and, hence, identification of an individual variant is by its names. In contrast, records in S-NET have anonymous variants containing named fields. Thus, variant identification is done primarily by analysis of the field set. As a consequence, input types of type signatures must be complete to ensure proper variant identification. Completeness may, for example, be achieved by the use of binding tags, which effectively mimick the named variants of conventional languages.

Whilst the conventional approach completely avoids the variant ambiguity, it also precludes subtyping on the basis of field names only. For instance, in our

example of type `body`, conventional subtyping would preclude the construction of a generic box that alters the body position expressed in terms of fields `x1` and `y1` that are common to all variants. As a result a box would require variants to be identified individually and specifically, even when the processing of fields `x1` and `y1` is the same for all variants, for instance, a coordinate shift `x1->x1+a`, `y1->y1+b`. The conventional OOP approach to this would be via a base class with fields `x1` and `y1` and a method `shift` to be inherited by all subclasses. This restricts the design in that there can be more than one set of common fields (which would require multiple inheritance), but more importantly since the significance of a common group of fields may become apparent only when an entirely new processing box is introduced into a streaming network, and in that case a re-design of the class hierarchy may become necessary.

Subtyping by subsetting as introduced in the beginning of this section does allow *a-posteriori* introduction of a supertype (equivalent to a base class), which obviates the re-design. The price to pay in implementation is the price of a run-time coercion (i.e. a selective copy of fields, or an extra level of indirection to avoid the need to copy), since it can no longer be assumed that the fields to be processed are necessarily a prefix of the field list.

### 2.5 Flow inheritence

Streaming networks promote pipelining whereby a record travels along a chain of boxes that apply various processing algorithms to its content. Since a box can legally be fed with a subtype of the input type, this would result in the loss of all fields that are not required by the input type, but these fields could possibly be required by another box further down the pipeline. To remedy that, the following type rule is introduced:

**Definition 3 (flow inheritance).**
*Let* $v^{[i]} \to \tau^{[i]}$*,* $i \in [1,\dots,n]$*, be the type signature of a box* $X$*. Furthermore, let each output type* $\tau^{[i]}$ *have* $m_i$ *variants* $\tau^{[i]} = \{w_1^{[i]},\dots,w_{m_i}^{[i]}\}$*. Then for any* $k \leq n$ *and any field or non-binding tag* $\phi \notin v^{[k]}$ *such that*

$$(\forall i \neq k) BT(v^{[k]}) \neq BT(v^{[i]}) \ \vee\ v^{[k]} \cup \{\phi\} \not\subseteq v^{[i]}\ ,$$

*the box* $X$ *can be subtyped* by flow inheritance *to the type* $X' : V^{[i]} \to T^{[i]}$*, where*

$$V^{[i]} = \begin{cases} v^{[i]} & \text{if } i \neq k, \\ v^{[k]} \cup \{\phi\} & \text{otherwise}; \end{cases}$$

*and*

$$T^{[i]} = \begin{cases} \tau^{[i]} & \text{if } i \neq k, \\ \tau_* & \text{otherwise}. \end{cases}$$

*Here* $\tau_* = \{V_1,\dots,V_{m_k}\}$ *and each* $V_i = w_i^{[k]} \cup \{\phi\}$*.*

Informally, an input variant can be extended with a new field $\phi$ (which can be a non-binding tag but not a binding tag), if it does not clash with any other variant. The output type associated with this input variant is extended with the field named $\phi$ in each of its variants unless it is present there already. Any number of flow inheritance extensions can be applied to a box, resulting in several fields being added. Value-wise, the extension is in terms of copying the value of the input record field $\phi$ over to the output record field with the same name[1]. If the output already contains an identically named field, then that field's value supersedes the inherited one. For convenience, we shall write box signatures in the form $(n, m)v^{[i]} \rightarrow w_j^{[i]}$, which signifies a box with input variants $v^{[i]}$ and the corresponding output types $\tau^{[i]} = \{w_1^{[i]}, \ldots, w_{m_i}^{[i]}\}$, $i \in \{1, \ldots, n\}$. Note that $n$ is a scalar integer that describes the number of input variants, whereas $m$ denotes a vector of $n$ integers describing the number of variants in each output type associated with one input variant.

Flow inheritance creates a subtyping hierarchy for boxes. For example, a box that accepts records with a single field named $x$ and which produces records with a single field name $y$ is a supertype of a box that accepts $\{x, z\}$ and returns $\{y, z\}$. As a side effect, flow inheritance can be a source of redundancy in type signatures. Indeed, in the above example if the signature of the *same* box contains the rules $\{x\} \rightarrow \{y\}$ and $\{x, a\} \rightarrow \{y, a\}$, then clearly the second rule can be deleted without changing the effective box type. Value-wise, the second rule carries additional information, namely that a record $\{x, a\}$ if presented to the input, will cause a record $\{y, a\}$ to appear with a potentially *different value* of $a$, while, assuming that $b$ does not occur anywhere in the signature, if $\{x, b\}$ is presented at the input it would cause the output of $\{y, b\}$ with the output value of $b$ being exactly the same as its input value. Still, as far as types are concerned, we can always assume that the signature is nonredundant, since the redundant rules change nothing in the type transformation defined by it.

## 3 S-Net box definitions

### 3.1 Box declarations

S-Net essentially is a language for specifying hierarchical networks of boxes statically interconnected by typed streams. As a pure coordination language S-Net does not provide any means for the specification of computations, i.e. the concrete behaviour of boxes. This is left to existing computation or box languages like C or SaC. Fig. 3 provides a definition of core S-Net syntax.

A box in S-Net is declared by the key word `box` followed by the box name. We distinguish two different kinds of boxes: *atomic boxes* and *compound boxes.* An atomic box is implemented using a compute language (as opposed to S-Net as a coordination language) and its operational behaviour is opaque to S-Net.

[1] Obviously, an implementation is free to simply switch references.

```
SNetBox       ⇒ AtomicBox | CompoundBox
AtomicBox     ⇒ box BoxName ( BoxSignature ) Body
BoxSignature  ⇒ Variant -> Type
Body          ⇒ { <<< LanguageName | LanguageCode >>> }
              | ;
CompoundBox   ⇒ box BoxName [ { [SNetBox ]+ } ] Connect
Connect       ⇒ connect SNetExpr
SNetExpr      ⇒ BoxName | Primitive | Sync
              | Combinator | Operator
              | ( SNetExpr )
```

**Fig. 3.** Grammar of S-Net box specifications

In contrast, compound boxes consist of recursively nested further box specifications (both atomic and compound) and a representation of their interconnection topology.

The extensional behaviour of an atomic box is specified in terms of a type signature that is restricted to a single mapping between input variant and output type. The specification of the intensional behaviour of a box is outside the scope of S-Net. In general, the code for an atomic box will be found in a separate file, but for convenience S-Net allows the programmer to inline foreign language code. This code is separated from S-Net code by the separator symbols `<<<` and `>>>`. S-Net does not process nor even parse the code in between these separator symbols. Instead, the code is passed on to the appropriate compiler. For S-Net to know which compiler to take, the foreign language code section starts with a language identification symbol, which is separated from the code by a bar symbol. Site-specific data like the exact compiler name, compiler flags, etc., are extracted from an S-Net configuration file using the language identifier.

Compound boxes consist of a potentially empty sequence of subbox specifications enclosed in braces. Unlike atomic boxes, compound boxes come without a type signature. Type signatures of compound boxes are inferred by the S-Net compiler. The specification of a compound box is completed by the definition of the interconnection topology of the local boxes following the key word `connect`. In S-Net we specify interconnection topologies by an expression language. These S-Net-expressions are made up of instances of the local boxes referred to by their name, a set of primitive boxes, a special synchronisation cell as well as a number of network combinators and network operators.

Using the geometric term “box” for the constituents of S-Net programs already insinuates a graphical or visual representation of S-Net code equivalent to the textual representation, we have outlined so far. Fig. 4 sketches out graphical representations of both atomic (left) and compound (right) boxes. Both contain the box name in a kind of status line, followed by their type signatures. Given the fact that providing a type signature is optional for compound boxes, this

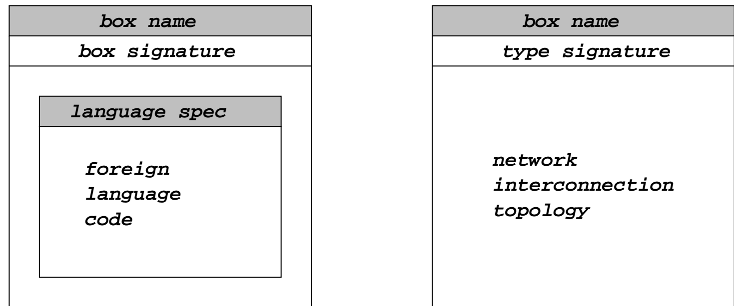


**Fig. 4.** Graphical representation of S-NET boxes

field may well be empty. Graphical representations of partially compiled S-NET programs may feature the inferred type signature here.

In the case of an atomic box the remaining field may remain blank. Otherwise, it contains a subbox with the inlined foreign language code and the language specification in the status line of the subbox. In the case of a compound box the remaining field shows a graphical representation of the interconnection topology. Their building blocks, both textual and graphical, are subject to the following sections.

### 3.2 Primitive boxes

Fig. 5 shows the concrete syntax of the three primitive boxes: *driver*, *link* and *plug*.

- The driver `[-` produces a stream of empty records. Unlike all other boxes in S-NET, it generates output without being triggered by an incoming record. The driver has the type signature { `->` {{}}}.
- The plug `-]` consumes any incoming record and produces absolutely nothing. The type signature of the plug is {{} `->` {}}.
- The link `--` realises a simple identity function: it forwards any incoming record to its output stream. The link has the type signature {{} `->` {{}}}.

| | | |
|---|---|---|
| *Primitive* | ⇒ | *Driver* \| *Plug* \| *Link* |
| *Driver* | ⇒ | `[` *[* `<` *TagName* `>` *[* `,` `<` *TagName* `>` *]** *]* `-` |
| *Plug* | ⇒ | `-` *[* `<` *TagName* `>` *[* `,` `<` *TagName* `>` *]** *]* `]` |
| *Link* | ⇒ | `-` *[* `<` *TagName* `>` *[* `,` `<` *TagName* `>` *]** *]* `-` |

**Fig. 5.** Grammar of S-NET primitive boxes

All three primitive boxes in S-NET can be refined by a set of tags *tags*. In the case of the driver this results in an output stream of records containing the given tags; the type signature becomes { -> {{*tags*}}}. For both the plug and the link the specification of additional tags leeds to a restriction of the input type; the type signatures become {{*tags*} -> {}} for the plug and {{*tags*} -> {{*tags*}}} for the link, respectively.

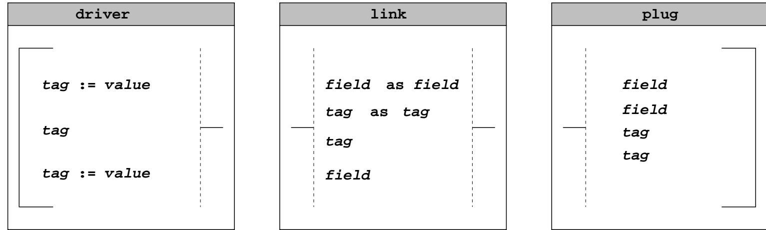


**Fig. 6.** Graphical representation of S-NET primitive boxes

Fig. 6 sketches out graphical representations for driver, link and plug. Each can be represented by a box with the respective symbol in the status line. If further specifications, e.g. tags, are present, they appear in the main field. Otherwise, it remains blank.

### 3.3 The synchrocell

The synchronisation cell, or synchrocell for short, is the only "stateful" box in S-NET; its concrete syntax is given in Fig. 7. A synchrocell starts with the key word `sync` followed by two or more variant specifications enclosed in brackets. Its behaviour can be refined by the use of tag specifications as explained in the sequel.

A synchrocell has storage for exactly one record of each given variant. When a record matching one of the variants arrives at the synchrocell, it is kept in this storage. Any record arriving thereafter that matches the same variant is immediatetely passed through the synchrocell. If the optional overflow flag is specified, that tag is added to each record passed through the synchrocell in this way. Let us assume all but one variants have been matched by an incoming record with the records properly stored in the synchrocell. As soon as a record arrives that matches the last remaining previously unmatched variant, all stored records are released. The output record is created by merging the fields of all stored records into the last matching record. This requires the various patterns to be pairwise disjoint. Otherwise, we had indistinguishable fields in the output record. If this prerequisite is not met right away, the key word `as` allows us to rename fields. As a consequence we may use patterns with overlapping field names as long as the fields are properly renamed to have unique names in the

| | | |
|---|---|---|
| *Sync* | ⇒ | **sync** ( *VariantPattern* ) *SyncTags* |
| *VariantPattern* | ⇒ | *Pattern* [ **,** *Pattern* ]* |
| *Pattern* | ⇒ | { *PatternField* [ **,** *PatternField* ] } |
| *PatternField* | ⇒ | *FieldName* [ **as** *FieldName* ] |
| | \| | < *TagName* > [ **as** < *TagName* > ] |
| *SyncTags* | ⇒ | [ *OutTag* ] [ *OverflowTag* ] [ *FixedPointTag* ] |
| *OutTag* | ⇒ | **out** < *TagName* > |
| *OverflowTag* | ⇒ | **ofl** < *TagName* > |
| *FixedPointTag* | ⇒ | **fix** < *TagName* > |

**Fig. 7.** Grammar of S-Net synchrocells

output stream. If the optional output tag is specified, that tag is also added to the output record.

If an incoming record matches more than one variant, all matching variants are marked as being matched and the incoming record is associated with all of them.[2] If an incoming record matches several variants, some of which are so far unmatched while others have already been matched by earlier records, the incoming record is only stored with the previously unmatched variants. However, given the fact that the eventual output record is constructed by merging the fields of all records involved, the sequence in which records arrive is irrelevant in this situation. Should an incoming record match all variants and the synchrocell is still in its initial state, i.e. all variants being unmatched, the record is immediately passed to the output stream. If present, the specified fixed point tag is added to the record in this case.

Once a synchrocell has received incoming records for each of its input variants, its purpose is fulfilled and the cell effectively dies. More precisely, all records received after a full match are immediately passed to the output channel with the overflow tag added where appropriate.

The synchrocell has a certain behaviour under flow inheritence. If a synchrocell stores a matching input record, it produces no output in response to this record. Hence, excess record fields, which would bypass the synchrocell otherwise, are discarded. Any record output after successful synchronisation is extended by the excess fields of the last incoming record because the synchrocell produces this output as a response to the input of this record. Last but not least, if a record is passed through the synchrocell in the case of overflow, there is output in response to input and, therefore, the excess fields bypass the synchrocell as usual.

Fig. 8 shows the graphical representation of a synchrocell. Following the name "sync" in the status line, the main field contains the patterns line by line. The double line to the right of the patterns insinuates the synchronisation.

[2] Of course, an implementation avoids copying the record.

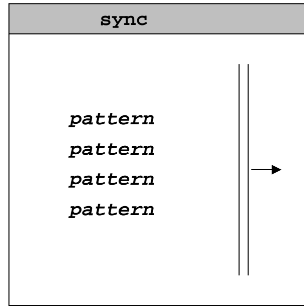


**Fig. 8.** Graphical representation of S-NET synchrocells

### 3.4 Network combinators

Complex network topologies are formed using the four network combinators: *serial*, *closure*, *choice* and *splitter*. Their grammar is defined in Fig. 9 while Fig. 10 sketches out the corresponding graphical representations.

*Combinator* ⇒ *Serial* | *Closure* | *Choice* | *Splitter*
*Serial* ⇒ *SNetExpr* `..` *SNetExpr*
*Closure* ⇒ *SNetExpr* `*`
*Choice* ⇒ *SNetExpr* `||` *SNetExpr*
  | *SNetExpr* `|` *SNetExpr*
*Splitter* ⇒ *SNetExpr* `!!` < *TagName* > [ `:` < *TagName* > ]
  | *SNetExpr* `!` < *TagName* > [ `:` < *TagName* > ]

**Fig. 9.** Grammar of S-NET network combinators

The binary serial combinator "`..`" connects the output of the left operand to the input of the right operand. The input of the left operand and the output of the right one become those of the resulting network. The serial combinator establishes computational pipelines. Graphically, it is represented by an arrow connecting the two argument boxes.

The unary closure combinator "`*`" (conceptually) replicates the operand infinitely many times and connects the replicas by the serial combinator. If the type signature of the operand contains a fixed point rule of the form `{<v>}->{<v>}` for some tag `v`, dynamic replication of the operand stops as soon as all records carry that tag. Otherwise, replication unfolds infinitely and no records can leave the closure, i.e., type-wise the closure becomes a plug. Implementation-wise the closure combinator realises a feedback loop, and the output stream of the operand

network is directly connected back to its input stream. This property is emphasised by the graphical representation in Fig. 10.

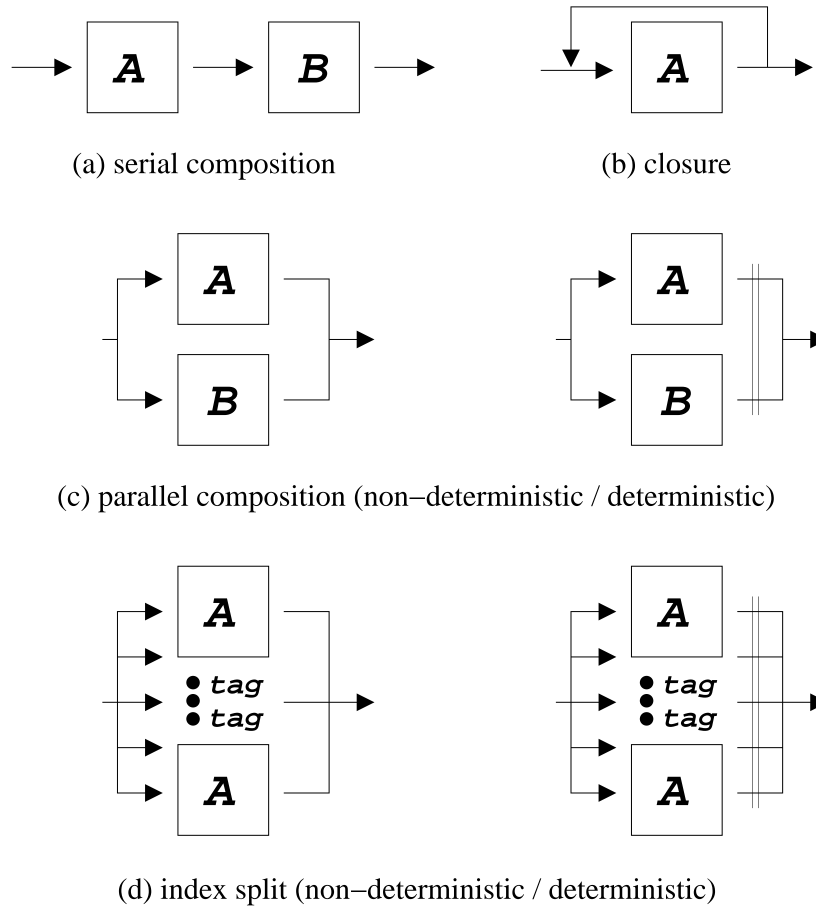


**Fig. 10.** Graphical representation of S-NET network combinators

The binary choice combinator "`||`" or "`|`" combines its operands in parallel. If an incoming record matches the type signature of one of the operand networks, it is sent to that network. Should an incoming record match the type signatures of both operand networks, it is non-deterministically sent to one of them. An implementation is free to choose an appropriate scheduling technique in this case. For example, it may send the record to the less loaded subnetwork for proper workload balancing. The graphical representation of the choice combinator employs a split and a merged arrow to visualise the choice operator.

The output streams of the operand networks are merged into a single stream which becomes the output stream of the combined network. Here, the two variants of the choice combinator, "`||`" and "`|`" behave differently. Whereas the "`||`" choice combinator merges the two output streams non-deterministically, the "`|`" variant preserves the sequence of records. More precisely, any output generated by one of the operand networks in response to an incoming record on the joint input stream is sent to the joint output stream before any records produced by any of the subnetworks in response to a subsequent input record. In the graphical representation an additional vertical double line symbolises the synchronisation necessary to achieve this deterministic behaviour.

Providing these two variants of the choice combinator is motivated by the observation that different application scenarios require different concrete operational behaviours of choice. The non-deterministic variant usually is more efficient since it allows the network to continue processing records as soon as they are available. However, in many situations it is crucial that a compound box behaves as an atomic box in that it preserves the sequence of causality and records may not overtake others. This comes at the price of holding back readily processed records from the output stream and waiting for other records to be sent first.

The binary index split combinator "`!!`" or "`!`" takes a subnetwork or box as its left operand and one or two tags as its right operand. Like the closure combinator, the index split combinator replicates the box operand, but connects the replicas using the choice operator. The number of replicas is conceptually infinite; each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the named tag, i.e., all records that have the same tag value will be processed by the same replica. If actually two tags are specified, the record is broadcast to all replicas with indices in the range between the value of the first tag and the value of the second tag.

The graphical representation of the index split combinator, as shown in Fig. 10, symbolically replicates the argument box with three vertical dots representing the dynamic number of replicas. The name(s) of the tag(s) that control replication are annotated at the vertical dots.

Analogously to the choice operator, the output streams of the replicas are merged into a single output stream of the combined network either non-deterministically ("`!!`") or under preservation of causality with respect to the sequence of records on the input stream. As in the case of the choice combinator an additional vertical double line visualises the necessary synchronisation in order to achieve this deterministic behaviour.

### 3.5 Network operators

Whereas the network combinators allow us to construct complex networks from simpler ones, the network operators can be used to manipulate the interface of a given network to the outside world. Fig. 11 shows the syntax of S-Net network operators.

```
Operator      ⇒  Pass | Strip | Set
Pass          ⇒  pass ( VariantPattern ) to SNetExpr
Strip         ⇒  strip ( Field [ , Field ]* ) from SNetExpr
Set           ⇒  set ( Parameter [ ; Parameter ]* ) in SNetExpr
Parameter     ⇒  FieldName := ExternalExpr
                 | < TagName > [ := Expr ]
Expr          ⇒  ExternalExpr | IntegerConst
ExternalExpr  ⇒  <<< LanguageName | ForeignExpr >>>
```

**Fig. 11.** Grammar of S-Net network specifications

The pass operator restricts the type signature of a subnetwork to certain patterns of incoming records (key word `to`) or outgoing records (key word `from`). Any incoming record must match one of the given patterns. Pattern matching is strictly by subset relationship on record fields. In contrast to the subtyping relationship, as defined in Section 2.2, there is no special treatment of binding tags in pattern matching. Matched record fields are sent to the subnetwork while excess fields are flow inherited by the output stream of the subnetwork. Using the key word `as` in the pattern specification, allows us to rename record fields before entering the subnetwork.

If the pass operator is used to filter the output stream of a subnetwork (key word `from`) only those fields matching one of the patterns actually leave the subnetwork whereas all others are discarded. Again, we may rename fields before they leave the subnetwork. The pass operator is a way to create separate name spaces for subnetworks.

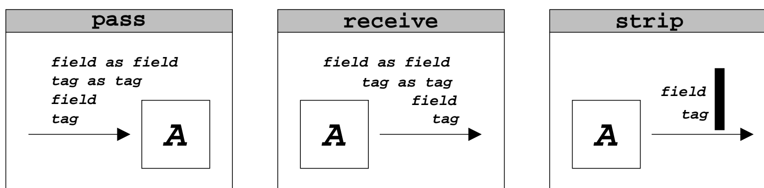


**Fig. 12.** Graphical representation of S-Net network operators

The strip operator resembles the second variant of the pass operator. It strips the named record fields from any record on the output stream and discards them. Whereas the pass operator allows us to specify which fields actually remain in outgoing records, the strip operator names those to discard.

Last but not least, the set operator provides a means to introduce initial field values to records. If an incoming record already contains a given field, its value

remains unaltered. Otherwise, the field is added an initialised with a value as specified in the set operator. For tags the intialisation value may simply be an integer constant. If it is left out, the tag is set to zero. However, for fields in general the situation is more complicated because S-Net as a pure coordination language deliberately lacks any notion of field value and the means to describe them. Hence, we follow the same path as with atomic box specifications and exploit the capabilities of a box language. Embraced within the separator symbols `<<<` and `>>>` we feature a box language identifier and separated by a single bar an expression in that langauge.

## 4 Illustrating example

— still missing —

## 5 Related work

The concept of stream processing has a long history. The view of a program as a set of processing blocks connected by a static network of channels goes back at least as far as Kahn's seminal work [2] and the language Lucid [3]. Kahn introduced the model of infinite-capacity, deterministic process network and proved that it had properties useful for parallel processing. Lucid was apparently the first language to introduce the basic idea of a block that transforms input sequences into output ones. A variable would represent such a sequence, acting as a stream of values of that variable in time. Ordinary operators in Lucid acted on variables point-wise, by effectively synchronising streams and applying the operation across pairs of corresponding stream elements. Additionally there were also some "temporal" operators, which were intended for altering the order of elements in a sequence.

Somewhat later, in the 80s, a whole host of synchronous dataflow languages sprouted, notably the languages Lustre [4] and Esterel[5], which introduced explicit recurrence relations over streams and further developed the concept of synchronous networks. These languages are still being used for programming reactive systems and signal processing algorithms today, including industrial applications such as the recent Airbus flight control system and various other aerospace applications [6]. The authors of Lustre broadened their work towards what they termed synchronous Kahn's networks[7, 8], i.e functional programs where the connection between functions, although expressed as lists, is in fact 'listless': as soon as a list element is produced, the consumer of the list is ready to process it, so that there is no queue and no memory management is required.

A nonfunctional interpretation of Kahn's networks is also receiving attention, the latest stream processing language of this category being, to the best of our knowledge, the MIT's StreamIt [9]. The latest comprehensive survey of stream processing and the underlying theory for it can be found in [10]. There is also a growing activity in *database stream processing* [11] , which concerns itself with the

problem of responding to a database query "on the fly", using the same limited-memory, sliding-window view of processing blocks that started with Lucid and continued through the aforementioned stream-processing languages. Still, despite much work having been done in various niche areas, stream processing has yet to be recognised as a general-purpose paradigm in the same sense as, for instance, object-oriented or functional programming.

Around the time that Lustre was introduced, David Turner[12] remarked that streams could be used as software glue for complex parallel software systems, even operating systems. In his interpretation, streams were lazy lists, which were produced on demand for their consumers. The lists were seen as an interface between the deterministic parts of a parallel system, which were pure stream-processing functions[3], and the external interleavers/mergers that realise the inter-process communication and capture its nondeterministic behaviour.

Besides these pragmatic considerations, we must mention here equally important theoretical advances in streaming networks. The key work in this area appears to have been done by Stefanescu, who has developed several semantic models for streaming networks starting from flowcharts [13] and recently including models for nondeterministic stream processing developed collaboratively with Broy [1]. This work aims to provide an algebraic language for denotational semantics of stream processing and as such is not focused on pragmatic issues. It nevertheless offers important structuring primitives, which are used as the basis for a network algebra (see [14]). It is interesting to note that apparently the StreamIt team [9] as well as ourselves [15] were unaware of those and had to re-invent them, albeit for purely pragmatic reasons.

## 6 Conclusion and future work

We have presented the basic language design of S-Net, a mostly functional typed stream processing language. S-Net allows us to specify component networks, which are interconnected by typed streams, in a declarative manner. Network topologies are effectively controled via subtyping on streams.

We are currently working on a prototype implementation of S-Net, which will feature C and SaC [16] as box languages. Given the implicitly data parallel capabilities of SaC [17], the combination of S-Net and SaC is particularly interesting as it allows the programmer to exploit both regular and irregular parallelism in a program. Still, with the clear separation between compute language and coordination language facilitates programming.

Further directions of future work include a graphical editor for S-Net specification and its integration with the S-Net compiler to form an integrated development environment. We are also interested to increase the number of supported box languages, in particular from the world of functional languages. A larger range of box languages would additionally give S-Net a potential role as glue for multi-language and multi-paradigm programming.

[3] but they could have been any self-contained procedures rather than pure functions, as long as the only access they had to each other's state was via stream communication

## Acknowledgements

This work was funded by the European Union IST-FET research project Æther. For more information on Æther, see `www.aether-ist.org`.

## References

1. Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science (2001) 99–129
2. Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, L., ed.: Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden, North-Holland (1974) 471–475
3. Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM **20** (1977) 519–526
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320
5. Berry, G., Gonthier., G.: The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
6. Binder, J.: Safety-critical software for aerospace systems. Aerospace America (2004) 26–27
7. Caspi, P., Pouzet, M.: Synchronous kahn networks. In Wexelblat, R.L., ed.: ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming. (1996) 226–238
8. Caspi, P., Pouzet, M.: A co-iterative characterization of synchronous stream functions. In Bart Jacobs, Larry Moss, H.R., Rutten, J., eds.: CMCS '98, First Workshop on Coalgebraic Methods in Computer Science Lisbon, Portugal, 28 - 29 March 1998. (1998) 1–21
9. Michael I. Gordon *et al*: A stream compiler for communication-exposed architectures. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. October 2002. (2002)
10. Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541
11. Babcock, B., et al.: Models and issues in data stream systems (invited paper). In: Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS 2002), Wisconsin, May 2002. (2002) 1–16
12. Turner, D.A.: An approach to functional operating systems. In Turner, D.A., ed.: Research topics in Functional Programming. Addison-Wesley University Of Texas At Austin Year Of Programming Series. Addison-Wesley Publishing Company (1990) 199–217
13. Stefanescu, G.: An algebraic theory of flowchart schemes. In Franchi-Zannettacci, P., ed.: Proceedings 11th Colloquium on Trees in Algebra and Programming, Nice, France, 1986. Volume LNCS 214., Springer-Verlag (1986) 60–73
14. Stefanescu, G.: Network Algebra. Springer-Verlag (2000)
15. Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: SNPD. (2003) 268–276
16. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming **13** (2003) 1005–1059
17. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401