

# S<sup>+</sup>NET: extending functional coordination with extra-functional semantics

R. Poss, M. Verstraaten, F. Penczek, C. Greck, R. Kirner, A. Shafarenko  
University of Amsterdam, The Netherlands  
University of Hertfordshire, United Kingdom

June 13, 2013

## Abstract

This technical report introduces S<sup>+</sup>NET, a compositional coordination language for streaming networks with extra-functional semantics. Compositionality simplifies the specification of complex parallel and distributed applications; extra-functional semantics allow the application designer to reason about *and* control resource usage, performance and fault handling. The key feature of S<sup>+</sup>NET is that functional and extra-functional semantics are defined orthogonally from each other. S<sup>+</sup>NET can be seen as a simultaneous simplification and extension of the existing coordination language S-NET, that gives control of extra-functional behavior to the S-NET programmer. S<sup>+</sup>NET can also be seen as a transitional research step between S-NET and AstraKahn, another coordination language currently being designed at the University of Hertfordshire. In contrast with AstraKahn which constitutes a re-design from the ground up, S<sup>+</sup>NET preserves the basic operational semantics of S-NET and thus provides an incremental introduction of extra-functional control in an existing language.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Functional specifications</b>	<b>4</b>
2.1	Notations . . . . .	4
2.2	Primitive networks . . . . .	5
2.3	Basic operational model for state management . . . . .	7
2.4	Functional combinators and composite networks . . . . .	7
<b>3</b>	<b>Transducer language</b>	<b>11</b>
3.1	Overview . . . . .	12
3.2	Specification language . . . . .	12
<b>4</b>	<b>Extra-functional specifications</b>	<b>14</b>
4.1	Overview . . . . .	15
4.2	Replication selection . . . . .	16
4.3	Identifiers for run-time activities . . . . .	17
4.4	Network labeling and selection . . . . .	18
4.5	Environmental exception handling . . . . .	18
4.6	Extra-functional isolation . . . . .	19
4.7	Extra-functional budget . . . . .	20
4.8	Projections: mapping specifications into processing agents . . . . .	20
4.9	Hardware affinity and mapping . . . . .	22
4.10	Environmental awareness . . . . .	23
4.11	Implementation services . . . . .	24
<b>5</b>	<b>Relationship to S-NET and design rationales</b>	<b>25</b>
5.1	Overview of functional changes to S-NET . . . . .	25
5.2	Stream connections and structural typing . . . . .	25
5.3	Usability of synchrocells . . . . .	26
5.4	Aggregate updates . . . . .	27
5.5	Identification of run-time activities . . . . .	28
5.6	Entity-centric vs. record-centric projections . . . . .	28
5.7	Lifetime of activities . . . . .	30
5.8	Environmental awareness . . . . .	31
<b>6</b>	<b>Summary and conclusions</b>	<b>32</b>
	<b>Acknowledgements</b>	<b>33</b>
	<b>References</b>	<b>33</b>

# 1 Introduction

S<sup>+</sup>NET provides a high level, declarative coordination language based on concepts borrowed from stream processing. S<sup>+</sup>NET is based on S-NET: a coordination language whose notation describes explicitly the data dependencies in a computation. As data movement is then exposed in the language semantics, mapping and managing the application to parallel platforms becomes simpler. S-NET is specified in [13, 10] and has been reported on in many published works, including [2, 4, 5, 3, 6, 9, 1, 8, 14, 15].

A key feature of S-NET is its focus on *declarative specifications*. The notation declares an intent of functional composition of primitive networks into more complex applications. The management of execution, including automatic parallelisation and automatic concurrent interleaving of activities, is fully delegated to a run-time system in software.

S<sup>+</sup>NET answers two general concerns that have been revealed through the past use of S-NET with industrial applications.

For one, it was believed originally that the S-NET technology ought to be equipped with intelligence to automatically optimize application execution, both during initial compilation and at run-time using feedback loops. One of the goals of the EU-funded ADVANCE<sup>1</sup> project was to attempt and demonstrate this. Unfortunately, one of the unescapable lessons re-learned during ADVANCE is that at any level of intelligence built into a system, there are some desirable optimizations that are *necessarily* out of reach from that system, although they could be reachable by letting a human author refine the specification manually [16].

Second, it was believed that the effectiveness of automated optimizations would most often be superior to the human skill, especially for large parallel systems, due to the complexity of the systems involved. In practice, we observed a more nuanced reality. S-NET has revealed across its use cases that each application has its own type of bottleneck, emerging as a consequence of both over-constraining in the specification and run-time factors in the implementation, such as hardware parameters or implementation quirks of the software run-time system [7]. Meanwhile, human operators equipped with high-level monitoring and profiling tools have proved well-able to understand these bottlenecks and subsequently restructure applications to avoid them, often *at a fraction of the cost that would be otherwise necessary to implement a new optimization* able to recognize and handle the bottlenecks automatically.

This situation has motivated a new perspective on coordination, which we attempt to capture with S<sup>+</sup>NET. On the one hand, general principles of software engineering make it desirable to keep the functional part of a specification devoid of operational semantics. This enables the automatic derivation of proofs of correctness (e.g. through the type system) and simplifies the mental model used programmers during the initial phase of specification. On the other hand, the coordination layer must provide tools to both *describe* the actual behavior of an implementation at run-time (inspectability), and *optionally prescribe* or constrain the run-time behavior after a human operator has determined that the additional prescriptions are desirable (adaptability). These requirements call for a two-layered system, where functional and extra-functional aspects of

---

<sup>1</sup><http://www.project-advance.eu>

an application co-exist side-by-side and can be used orthogonally from each other.

With this background requirement in place, the question remains of what tools to place in the extra-functional toolbox. Following the design guidelines of S-NET, S<sup>+</sup>NET strives to provide *composable* specification operators that each determine an orthogonal aspect of the system.

This following technical report thus describes S<sup>+</sup>NET and its extra-functional contributions to S-NET. In the process of defining S<sup>+</sup>NET, the authors also took the opportunity to recognize shortcomings in the functional part of S-NET; the functional core of S<sup>+</sup>NET is comparatively more simple and its primitives are more orthogonal than S-NET's.

The report is organized as follows. In section 2 we present the part of S<sup>+</sup>NET's dedicated to functional specifications. One of the new functional concepts introduced in S<sup>+</sup>NET is the "transducer" component, subsequently further described in section 3. In section 4 then presents S<sup>+</sup>NET's extra-functional constructs and their semantics. We finally provide a more in-depth discussion of the differences between S-NET and S<sup>+</sup>NET in section 5.

## 2 Functional specifications

The functional core of S<sup>+</sup>NET uses only two *primitive components* and defines component network composition using *network combinators* over them. The elementary components are *boxes* and *transducers*, described in section 2.2. Boxes abstract entire programs provided externally to S<sup>+</sup>NET. Transducers enable expressing simple computation and synchronization constructs within the S<sup>+</sup>NET language. Two binary combinators assemble heterogeneous composite networks, and three unary combinators define more complex functional behaviors for their network operand. The combinators are detailed in section 2.4.

The interconnect between components appears as if each component had a single input stream and output stream, that is, the input/output events of a single component are observed in some total order. Each event on such a stream is called a *record*. Records are represented as sets of label-value pairs. Labels are called *field names* and values are called *fields*. Fields are either integer *scalars*, or *references* to data structures from the box language domain. References are opaque to coordination, although S<sup>+</sup>NET and box languages must cooperate to marshall data to and from streams of bytes suitable for communication over network channels.

### 2.1 Notations

We present the composition constructs of S<sup>+</sup>NET in the following sections.

As in S-NET, record types are noted using curly brackets around the set of field names, for example {a,b} which is equivalent to {b,a}. Furthermore, S<sup>+</sup>NET standardizes the following notations:

- *source notation*, e.g. "A .B" which provides a standard syntax to enter specifications into the S<sup>+</sup>NET system;
- *algebraic notation*, e.g. C(A, B), which provides a symbolic representation of input specifications, or specifications after partial automatic transformation by a S<sup>+</sup>NET implementation;

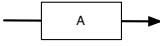

Name	Graphical representation	Algebraic	Source notation
Box		$\text{Box}(A)$ $B \rightarrow \mathcal{N}$	<code>box A ...</code>
Transducer		$\text{Transduce}(S)$ $S \rightarrow \mathcal{N}$	<code>[ S ]</code>

Table 1: Notations for S<sup>+</sup>NET’s primitive networks.

- *graphical notation*, to visualize specifications and provide a high-level intuition.

The examples hereafter use either algebraic, source or graphical notation, depending on which best illustrates a concept at hand.

## 2.2 Primitive networks

S<sup>+</sup>NET provides two primitive networks, from which all specifications are derived: *boxes*, which capture components developed externally, and *transducers*, which are stateful boxes whose behavior is implemented using a simple expression language within S<sup>+</sup>NET. The corresponding notations are given in table 1.

### 2.2.1 Boxes: stateless transformations

The main elementary functional construct is the encapsulation of an entire program, called component or *box*, able to work asynchronously on a stream of input and producing a stream of output.

Both imperative and declarative languages qualify as box implementation languages. The box language infrastructure around each component must offer clear primitives to set up a box, provide it with input, save or checkpoint its management state (e.g. heap managers, debugging metadata) and tear down the environment. A box implementation should be resource-agnostic, that is, able to perform its transformation regardless of the hardware resources selected for it by the S<sup>+</sup>NET coordination layer. In its simplest form, a box encapsulates a side-effect free, stateless function implemented in a standard programming language like C or C++.

A box can be described as a function from its input stream to its output stream. Boxes must appear functionally pure, that is, the output from one input record can be fully computed using that input record only, and a box terminates after it produces its last output. A box is characterized by a *box signature*: a mapping from an input type to a disjunction of output types. For example,

```
box foo ((a, b) -> (c) | (c,d));
```

declares a box `foo` that expects records with two fields labelled `a` and `b`. The box, when activated, responds with zero or more records that either have only one field labelled `c`, or two fields labelled `c` and `d`.

The set of field names naturally induces a *type signature* for every stream-to-stream transformation<sup>2</sup>. Type compatibility and subtyping are determined by

<sup>2</sup> General type signatures use set notation for record types, with curly brackets. For box signatures, order matters: the box implementation might only support positional arguments and record fields are then provided at run-time in the order specified. This is why box signatures use parentheses instead of curly brackets.

set inclusion. For example, the box `foo` above would accept a record with type  $\{b,a,c\}$  as input, but not  $\{a\}$  nor  $\{b,c,e\}$ . Subtyping on input types of boxes raises the question of what happens to the excess fields. S<sup>+</sup>NET defines *flow inheritance*, whereby excess fields from incoming records are not just ignored in the input record of a network entity, but are also attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance prove to be indispensable features when it comes to combine boxes designed in isolation into a larger application.

### 2.2.2 Transducers: stateful, finite synchronizers

A box, already described in the previous section, is stateless and can only split a record into parts. The transducer construct expresses the complementary operation: merging two or more records into one, possibly performing computations on them. Transducers are defined as finite state machines<sup>3</sup>; the number of different states is kept finite so that typing and correctness can be decided statically, and so that liveness and future state behavior can be predicted at run-time.

All specifications for transducers specify a finite set of states, conditions for transitions between them, and actions to carry out upon state changes. Actions include capturing (part of) the input using *hold variables* and/or constructing records to emit on the output stream. Hold variables can capture one record each, and have full/empty semantics: they can only be assigned when empty, and read or reset to the empty state when full.

Transitions can be conditional on input records, but conditions cannot involve previously captured input. Consequently, the type signature of the transducer can be determined statically for every state, as well as whether the hold variables stay consistent, i.e. each hold variable is not set when full and not reset when empty.

Any input record that arrives to a transducer and not accepted by a transition from the current state is output, unchanged, by the transducer. A transducer is said to be *inactive* whenever it is currently in its initial state and all its hold variables are empty; it *terminates* when it reaches a state with no outgoing transition.

More details on the transducer sub-language are given in section 3.

### 2.2.3 Multiple input, multiple output and tags

A distinguishing feature of S<sup>+</sup>NET is that it neither introduces streams as explicit objects nor does it define network connectivity through explicit wiring. Instead, it uses algebraic formulae, described below in section 2.4, to compose streaming networks. The restriction of type signatures to a single logical input and a single logical output stream (SISO) is essential for this. However, this is not to say that S<sup>+</sup>NET actually implements logical streams using a single communication channel; MIMO specifications are also possible. For this, records may be additionally be marked using a single *tag*, noted between angle brackets, for example  $\langle t \rangle$ . The inclusion of a tag in a component signature changes the type equivalence to only match records with exactly the same tag. Therefore

---

<sup>3</sup>theoretical construct of the same name, described in [12, Chap. 4], of which they are a practical application.

tags in one component's output can only be matched to components with the same tags in their declared input type, and an implementation can route this communication through dedicated channels. Tags may also be used as scalars to distinguish between sub-streams.

### 2.3 Basic operational model for state management

Stateful constructs like the transducer evolve through a lifecycle at run-time. Other functional combinators define below also define implicit state. State necessarily occupies space at run-time and breaks referential transparency, and thus defines *objects* at run-time that must be managed.

The basic properties of these objects are defined below; the rest of the S<sup>+</sup>NET specification refers to these properties and determines how they are managed by a run-time system.

To start, any stateful construct goes through the following phases:

1. when initially instantiated, it is *inactive*;
2. during its lifetime, it may go through one or more *activity cycles*, during which it is stateful, interleaved by inactivity periods;
3. it may reach a state past which it either always behave like the identity function, or never processes input records again, at which point it is said to be *terminated*.

A (sub-)network is said to be *live* as long as it is not terminated. Note that a network can be both active and terminated. For example, a transducer is known to be terminated whenever it reaches a state with no possible transition. However, at that point it may still have non-empty hold variables. In accordance to previous work in process management, we call *dead* those networks which are both terminated and inactive; and *zombie* those which are terminated but still active (retain state).

When proven at run-time, the death property enables the simplification of the process network by eliminating replicas or run-time state that have become unnecessary, i.e. *garbage collection* [1]. Depending on the application domain, the implementation may also permit automatic garbage collection of zombie networks.

In this context, the lifecycle of primitive networks deserves attention. It is possible to consider a box as a network that starts inactive, then becomes repeatedly active for each successive input record. Because it is stateless, it is also possible to consider it as a network which terminates after each successive input record, then instantiated anew for the next input record. Similarly, whenever a live transducer becomes inactive, either it can be re-activated for the next input record or terminated and reinstantiated anew. This duality is functionally neutral; however, which approach is taken in an implementation has extra-functional consequences on latency and throughput, and is thus observable and controllable in S<sup>+</sup>NET, via the use of projections (cf. sections 4.8 and 5.6).

### 2.4 Functional combinators and composite networks

S<sup>+</sup>NET provides the following functional combinators, which compose networks:

- finite *composition* of two or more networks, to define pipelines;


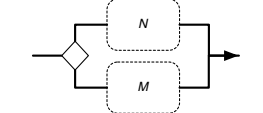
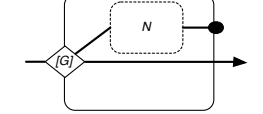
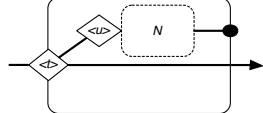
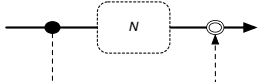
Name	Graphical representation	Algebraic	Source notation
Composition		$C(\dots)$ , eg. $C(N, M)$ $\mathcal{N}^* \rightarrow \mathcal{N}$	$N . M$ “dot dot”
Selection		$S(\dots)$ , eg. $S(N, M)$ $\mathcal{N}^* \rightarrow \mathcal{N}$	$N   M$ “or”
Ordered replication composition		$C_G^*(N)$ $\mathcal{G} \times \mathcal{N} \rightarrow \mathcal{N}$	$N * G$ “star”
Unordered replication composition		$C_{\langle t \rangle, \langle u \rangle}^!(N)$ $\mathcal{T}^2 \times \mathcal{N} \rightarrow \mathcal{N}$	$N ! * \langle t \rangle \langle u \rangle$ “blink star”
Reordering		$R(N)$ $\mathcal{N} \rightarrow \mathcal{N}$	$?N\#$

Table 2: Notations for S<sup>+</sup>NET’s functional combinators.

- finite *selection* between two or more networks, to define routing between alternative processing pipelines;
- arbitrarily deep *ordered replication composition*, which dynamically replicates a network and composes the replicas in a deterministically ordered pipeline;
- arbitrarily deep *unordered replication composition*, which dynamically replicates a network and composes the replicas in a non-deterministically ordered pipeline.

Both simple selection and the replication operators introduce processing concurrency, that is, non-determinism in the order records are visible in the logical output stream. To force reordering when needed, S<sup>+</sup>NET also provides a *re-ordering* combinator which ensures that output records are visible in the same order as the input record that caused them.

The corresponding notations are given in table 2. The following sections introduces each combinator in more details.

#### 2.4.1 Simple composition and selection

Composition ( $C(A, B)$ , or “A . B” in source) constructs a new network where the logical streams are connected in series. The composite network can be thought as performing the function  $B \circ A$  concurrently over each input record to produce output records. This form of composition preserves ordering of the input stream in the output stream.

Selection ( $S(A, B)$ , or “A | B” in source) constructs a new network where each input record is routed to one of the branches. Which route is taken is determined by best match on the input type of the alternative networks. The composite



network can be thus thought as performing a type-based functional choice between  $A$  or  $B$ . Moreover, selection introduces stream concurrency between the alternatives: when two successive records  $r_a$  and  $r_b$  are presented on the input stream, to be routed to  $A$  and  $B$ ,  $B$ 's output for  $r_b$  may appear interleaved in any way with  $A$ 's output for  $r_a$  on the output stream. Ambiguous selections are resolved in the specification order<sup>4</sup>. For example with  $S(A, B)$ , if  $A$  accepts  $\{a\}$  and  $B$  accepts  $\{b\}$ , a record with type  $\{a, b\}$  will be routed to  $A$ .

At any point during execution, a selection is active if at least one of its branches is active; when it is terminated, it terminates also all the branches. We define by extension the *dynamic liveness arity* and *dynamic activity arity* of a composite, which is the current number of live and active instances of the inner networks at run-time, respectively. Dynamic liveness arity is intuitively associated with passive spatial complexity, implied by the cost of maintaining the network instances in the run-time environment; dynamic activity arity is intuitively associated with active spatial complexity, implied by the cost of actual accesses to the instances' state.

The logical output stream of the composite network is the fusion of the concurrent sub-streams after processing by the replicas. As with simple selection, fusion is non-deterministic and sub-streams may appear interleaved.

### 2.4.2 Reordering

If the non-determinism in the output order of selection is not desired, it is possible to encapsulate a network  $N$  in the reordering combinator which reintroduces the input order, noted  $R(N)$ . For example, with  $R \circ S(A, B)$ , or “ $?A|B\#$ ” in source form, when presented with successive records  $r_a$  and  $r_b$ ,  $A$ 's last output for  $r_a$  will be observed on the output stream of the composite network before  $B$ 's first output for  $r_b$ .

At any point during execution  $R(N)$  is active if either  $N$  is currently active or if the composite network is currently holding records for reordering. It terminates  $N$  when terminated itself.

### 2.4.3 Ordered replication composition

The ordered replication composition over  $N$ , noted  $N!G$  in source form or  $C_G^*(N)$  in algebraic form, defines a composite network whose functional semantics are defined as follows:

- $G$  is a *guard pattern*, which expresses record types together with an optional field predicate;
- it processes records through an ordered composition of replicas of  $N$  as long as records do *not* match the guard pattern.

In other words, ordered replication composition can be thought of defining an infinitely enumerable set of replicas  $\{[N]_i \mid i \in \mathbb{N}\}$  and processing each input record via the functional definition

$$[C_G^*(N)] = C_G^*([N]_0)$$

$$C_G^*([N]_i)(r) = \begin{cases} r & \text{if } r \text{ matches } G \\ C([N]_i, C_G^*([N]_{i+1}))(r) & \text{otherwise} \end{cases}$$

<sup>4</sup> This selection determinism is a divergence from the approach taken with S-NET's “parallel composition” construct; we discuss this in sections 4.2 and 5.

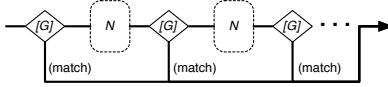


Figure 1: Dynamic unfolding of  $C_G^*(N)$ .

The corresponding dynamic unfolding is illustrated in fig. 1; it implements guarded functional recursion. This is the alternative proposed by S<sup>+</sup>NET to cycles in specification graphs to define repeated behavior.

Like with selection, activity is defined inductively from the corresponding state of the inner replicas, and termination of the composition implies termination of the replicas. Dynamic liveness and activity arities are also defined transparently. In addition, replication composition defines *dynamic depth* to be the number replicas involved in the linear chain that are not yet terminated. Dynamic depth is intuitively associated with the time complexity of processing, since any terminated replica is neutral w.r.t composition.

An implementation is expected to remove terminated replicas dynamically from the composition chain, i.e. keep the dynamic liveness arity synchronized with the dynamic depth.

#### 2.4.4 Unordered replication composition

The unordered replication composition is intended to capture the non-deterministically ordered sequence of transformations on a shared data structures between computation agents. For example, it can be used to express the transformation of a graph data structure as an non-deterministically ordered set of concurrent subgraph-to-subgraph transformations. The intent is to let the S<sup>+</sup>NET implementation schedule the concurrent activities either in parallel using either locking or speculation and transactional storage to guarantee the linear order (cf. section 5.4).

The functional semantics are defined as follows. The construct  $N!*\langle r \rangle \langle p \rangle$  in source form or  $C_{\langle r \rangle, \langle p \rangle}^1(N)$  in algebraic form, defines a composite network. The first tag  $\langle r \rangle$  marks *constructor records* and the second tag  $\langle p \rangle$  marks *payload records*. Any contiguous sub-sequences of constructor records is called a *constructor sub-sequence*. A single constructor sub-sequence (of zero or more constructors) followed by one payload record is called a *processing sub-sequence*. Any record that are neither constructors or payloads, called *pass-through records*, are forwarded to the output stream unchanged.

In each processing sub-sequence of the input stream, the behavior is defined as follows:

- the constructors are collected in a list;
- if there are no constructors, the payload is passed through; otherwise
- one constructor is picked *at a non-deterministic position* of the list and removed from the list;
- the picked constructor and the payload are presented as input to a new (fresh) replica of the inner network;
- any constructor produced by the inner network is appended to the constructor list;

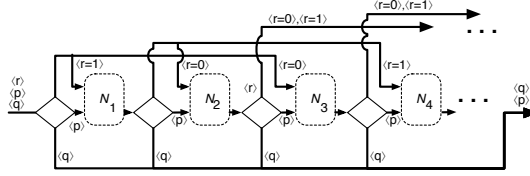


Figure 2: Dynamic unfolding of  $C_{(r),(p)}^!(N)$ .

- the resulting constructor list and the payload produced by the inner network are processed by a new instance of  $C_{(r),(p)}^!(N)$ ;
- the original replica is terminated.

An example dynamic unfolding is illustrated in fig. 2. In this example, each replica of the inner network accepts one  $\{r\}$  and one  $\{p\}$  record. Each replica also emits conditionally two new records  $\{r=0\}$  and  $\{r=1\}$  followed by a new  $\{p\}$  record, possibly interleaved with  $\{q\}$  records. When the processing sub-sequence  $\{r=0\}, \{r=1\}, \{p\}$  is presented on the input, the following happens. The first two  $\{r\}$  records are accumulated in a list. The 2nd constructor is picked first ( $\{r=1\}$ ), a first replica  $N_1$  is created, and both the constructor and  $\{p\}$  record are presented to  $N_1$ . The  $\{r\}$  records produced by  $N_1$  are then appended to the list, which becomes  $[\{r=0\}, \{r=0\}, \{r=1\}]$ . Again the 2nd constructor is picked first ( $\{r=0\}$  from  $N_1$ ), and defines a new replica  $N_2$ . The  $\{p\}$  record output by  $N_1$  is then sent to  $N_2$ . The  $\{r\}$  records produced by  $N_2$  are appended to the list, then the 1st element is picked from the remaining list ( $\{r=0\}$  from  $N_2$ ). And so on. Meanwhile, all the interleaved  $\{q\}$  records are also forwarded to the output stream.

All processing sub-sequences of the logical input stream are processed concurrently, by distinct processing sets of replicas. In each processing set, any given replica is only used once, over exactly one constructor and one payload record. The replica is terminated after it has processed the input payload and optionally output one new payload. A single top-level processing sequence is said to *complete processing* when all replicas in its processing set have terminated and no constructor records are left unprocessed.

As can be seen, this combinator combines two forms of non-determinism: both concurrency between input processing sequences and concurrency between constructor records at any level of the recursion, which non-deterministically change the composition order of the remaining replicas.

As with the previous forms of replication, activity is defined inductively from the corresponding state of replicas. When terminated, the replication terminates its replicas. Dynamic liveness and activity arities are defined as the total number of live/active replicas across all processing sets. The dynamic depth of the network is the maximum of the dynamic depths of the individual processing sets. As with ordered replication composition, an implementation is expected to remove terminated replicas automatically from the run-time environment.

### 3 Transducer language

A box, already described in the previous section, is stateless and can only split a record into parts. Moreover, a box is defined externally to S<sup>+</sup>NET and its

specification is not known at the level of S<sup>+</sup>NET. In contrast, transducers provide a way to define simple computations on records within the S<sup>+</sup>NET language itself. Like boxes, once defined, transducers behave as primitive networks (cf. section 2.2)

### 3.1 Overview

The transducer construct expresses the complementary operation: merging two or more records into one, possibly performing computations on them. Transducers are defined as finite state machines<sup>5</sup>; the number of different states is kept finite so that typing and correctness can be decided statically, and so that liveness and future state behavior can be predicted at run-time.

All specifications for transducers specify a finite set of states, conditions for transitions between them, and actions to carry out upon state changes. Actions include capturing (part of) the input using *hold variables* and/or constructing records to emit on the output stream. Hold variables can capture one record each, and have full/empty semantics: they can only be assigned when empty, and read or reset to the empty state when full.

Transitions can be conditional on input records, but conditions cannot involve previously captured input. Consequently, the type signature of the transducer can be determined statically for every state, as well as whether the hold variables stay consistent, i.e. each hold variable is not set when full and not reset when empty.

Any input record that arrives to a transducer and not accepted by a transition from the current state is output, unchanged, by the transducer. A transducer is said to be *inactive* whenever it is currently in its initial state and all its hold variables are empty; it *terminates* when it reaches a state with no outgoing transition.

### 3.2 Specification language

S<sup>+</sup>NET provides a comprehensive syntax for transducers, to keep the specification short in simple cases and to factor regular behavior. To start with, it is common to express housekeeping coordination tasks that are stateless, for example duplicating records or performing simple arithmetic on scalar fields. Transducers can be defined with only one state and no captured input as follows:

```
[| {a, b, c} -> [emit {a=input.a, z=input.a, t=0}; emit {b,
                a=input.b, c=input.c+1}] |]
```

This transducer consumes records of type {a,b,c} and for each input creates two new records. The first output record has field **a** with the original value, field **z** with the same value and a scalar **t** set to zero. The second record has fields **b** with the original value, **a** with the same value as **b** and the scalar **c** incremented by one. A stateless transducer is also called “filter”, and equivalent to the S-NET construct of the same name.

To capture input, a specification must define hold variables. For example, the transducer defined with

---

<sup>5</sup>Transducers are named after the theoretical construct of the same name, described in [12, Chap. 4], of which they are a practical application.

```
[| var x;
  start: {a} -> [x := input] ha; {b} -> [x := input] hb;
  ha: {b} -> [emit input + x; reset x] start;
  hb: {a} -> [emit input + x; reset x] start;
|]
```

has one hold variable  $x$ . When in the `start` state, it accepts either records of type  $\{a\}$  or  $\{b\}$ , captures them in  $x$  and changes to a state where it can accept a record of the other type, combine it (using set union) and output the result, before resetting to the initial state.

The record expression after the label must be an exact match on the input record. The labels must match, and a scalar value, if specified, must match the input's scalar as well; this uses first match if there are multiple guards in the same state. For example, `[| {a=9} -> [emit {a=0}]; {a} -> [emit {a=input.a+1}]; |]` specifies a filter that increments  $a$  modulo 10.

To enable inheritance, a transducer can accept records with more fields than are matched in the guard, as follows: `[| {a}+x -> [emit {a=input.a+1}+x] |]`. This specifies a filter that accepts any input with at least field  $a$  and produces as output a record with  $a$  incremented, together with the remaining fields of the input. By extension, a guard can omit the match pattern and only use a “remainder” part, for example: `[| a: x -> [emit x+{even=1}] b; b: x -> [emit x+{odd=1}] a; |]`. This transducer adds scalars `odd` or `even` to alternate input records, regardless of their type.

It is also possible to manipulate state labels as integer values within a fixed range, and use *finite-size arrays* of hold variables indexed by the state labels. For example:

```
[| label [0..3]; var h[3];
  n=0..2/ n: x -> [h[n] := x] n+1;
  3: x -> [emit union(h)+x; reset h] 0; |]
```

This specifies 4 states labeled from 0 to 3, and 3 hold variables. The transition prefix “`n=0..2/`” defines a *label iterator*  $n$  and causes the duplication of the remainder of the transition specification for all specified values of  $n$ . Therefore, this transducer merges every successive 4 records into one, regardless of type.

Label iterators can be used with finite-size arrays of guards, too. For example, the transducer

```
[| label [0..3]; var h[3]; guard t[3] = {a},{b},{c};
  n=0..3/ n: t[n] -> [h[n] := input] n+1;
  4: {d} -> [emit union(h)+input; reset h] 0; |]
```

merges subsequences of  $\{a\}$ ,  $\{b\}$ ,  $\{c\}$  and  $\{d\}$  records, exactly in that order.

Finally, it is also possible to express synchronization of multiple successive records in arbitrary order using label sets defined as finite-size arrays of bits, as illustrated by the example in fig. 3. This transducer declares an array of 3 state bits, i.e. 8 possible states. For every  $i \in \{0, 1, 2\}$ , “ $\sim s[i]$ ” matches any state of the label array where bit  $i$  is not set; i.e.  $\sim s[0]$  matches labels 000, 010, 100 and 110;  $\sim s[1]$  matches 000, 001, 100 and 101, and  $\sim s[2]$  matches 000, 001, 010 and 011. “ $t[i]$ ” then uses the corresponding guard and “ $h[i]$ ” the corresponding hold variable. At the end of the transition “ $s[i]$ ” sets bit  $i$  of the bit array relative to the actual state matched on the left hand side. In

```

[] guard t[3] = {a},{b},{c}; label s[3]; var h[3];
i=0..2 / ~s[i]: t[i] -> [h[i] := input] s[i];
s[0..2]: -> [emit union(h); reset h] ~s[0..2]; []

```

(a) Specification

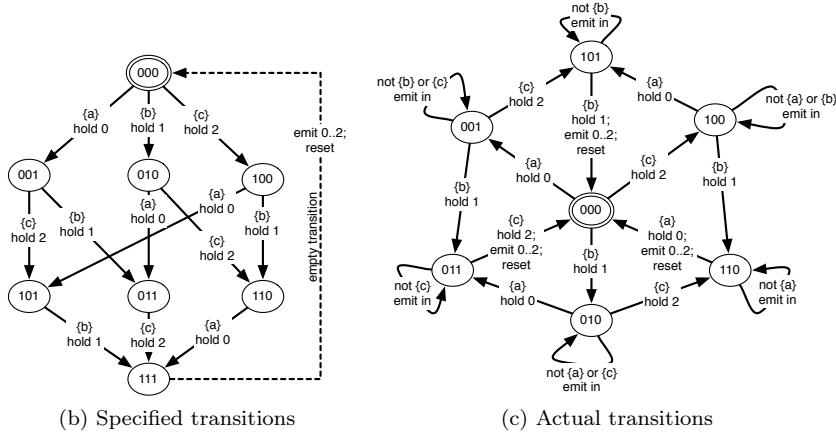


Figure 3: Transducer that merges triplets of  $\{a\}$ ,  $\{b\}$ ,  $\{c\}$  records in any order.

the second transition specification,  $s[0..2]$  applies to the single state where all bits in the array are set, i.e. label 111. It defines an *empty transition*, i.e. a transition whose action is taken as soon as its origin state is reached by another transition<sup>6</sup>. Its resulting state specification unset all bits, i.e. resets to label 000. As explained earlier, unspecified transitions cause the input record to be emitted as-is on the output stream.

This specific example can be reused to synchronize any finite set of record types by simply extending the size of the arrays; we found it so ubiquitous in concrete applications that S<sup>+</sup>NET proposes a simplified syntax for it: `[] {a},{b},{c} []`. This extends S-NET’s “synchrocell” concept, as discussed in section 5.3.

## 4 Extra-functional specifications

S<sup>+</sup>NET also provides *extra-functional combinators*, described in the following sub-sections, which can be layered on top of arbitrary networks. In contrast to functional combinators, the extra-functional combinators were designed so that adding them to a network does not influence its input-output value relationship.

More specifically, although they *can* influence value computations, the S<sup>+</sup>NET programmer should not *expect* the extra-functional combinators to yield the desired functional effects. This is because any particular implementation of S<sup>+</sup>NET may not support some of the extra-functional combinators and replace them with transparent constructs with no effect. For instance, the “environment awareness” enables a program to read values that describe the environment into a record’s tag, for example the number of cores in the resource where a sub-network is currently running. An application can make use of this number, but

<sup>6</sup>Of course, a cycle of empty transitions is not permitted, otherwise a transducer could define non-terminating actions.

Name	Graphical representation	Algebraic	Source notation
Replication selection		$S_{(c)p}^*(N)$ $\mathcal{T}^3 \times \mathcal{P} \times \mathcal{N} \rightarrow \mathcal{N}$	$N! \langle c \rangle p$
Labeling		$\alpha_X(N)$ $\mathcal{L} \times \mathcal{N} \rightarrow \mathcal{N}$	$N \cdot X$
Environmental exception handling		$\beta_{X(a=E)}(N)$ $\mathcal{L} \times \mathcal{E} \times \mathcal{F} \times \mathcal{N} \rightarrow \mathcal{N}$	$N\$X(a=E)$
Extra-functional isolation		$\theta_{X/R}(N)$ $\theta_{X/R}^+(N)$ $\mathcal{L} \times \mathcal{R} \times \mathcal{N} \rightarrow \mathcal{N}$	$N/X/R$ $N/X/+R$
Extra-functional budget		$\rho_{X:R}(N)$ $\mathcal{L} \times \mathcal{R} \times \mathcal{N} \rightarrow \mathcal{N}$	$N/X:R$
Projections to agents		$\gamma_X^e(N)$ $\gamma_X^r(N)$ $\tau_X^\omega(N)$ $\tau_X^\epsilon(N)$ $\mathcal{L} \times \mathcal{N} \rightarrow \mathcal{N}$	$N/X!ge$ $N/X!gr$ $N/X!te$ $N/X!to$
Resource affinity and assignment		$\phi_{X@Y:a}(N)$ $\mathcal{L}^2 \times \mathcal{P} \times \mathcal{N} \rightarrow \mathcal{N}$	$N/X@Y:a$
Environment awareness		$\delta_{\langle t \rangle, a}(E)$ $\mathcal{T} \times \mathcal{F} \times \mathcal{D} \rightarrow \mathcal{N}$	$[\langle t \rangle . a = E]$

Table 3: Notations for S<sup>+</sup>NET’s extra-functional primitive network and combinators.

acknowledge that a particular implementation may not support this combinator and always leave the record tag unmodified instead.

## 4.1 Overview

S<sup>+</sup>NET provides a new primitive network for environment feedback, noted  $\delta$ , and the following combinators:

- replication selection, noted S\*;
- network labeling, noted  $\alpha$ , used to designate sub-networks in the specification of other extra-functional combinators;
- environmental exception handling, noted  $\beta$ ;
- extra-functional requirements on isolation and budget, noted  $\theta$  and  $\rho$ ;
- projection into processing agents, noted  $\gamma$  and  $\tau$ ;
- hardware affinity and mapping, noted  $\phi$ .

The corresponding notations are given in table 3. All these constructs, except for  $\beta$ ,  $\delta$  and  $S^*$ , are functionally neutral: they do not change the input-output relationship of the encapsulated network.

Both  $\beta$  and  $\delta$  read and modify scalar values in records, and thus establish a bridge between functional and extra-functional semantics;  $S^*$  explicitly manipulates network replicas and thus can influence the functional semantics via output reordering and state management. However, we consider that an  $S^+$ NET specification is only well-formed if its input-output relationship stays valid for the application when all extra-functional combinators or  $\delta$  are elided.

## 4.2 Replication selection

The network combinator  $S^*$  expresses replication selection with optional extra-functional choice between replicas. This construct can be used to manually fine-tune the exploitation of parallel hardware; it is not intended for use during an initial specification or to when determining functional correctness of a specification.

The construct  $S_{\langle c \rangle p}^*(N)$  is a network specification if  $N$  is a network specification. It is also noted “ $N! \langle c \rangle p$ ” in source form.  $N$  is called the *inner network*;  $\langle c \rangle$  is the *selection tag* and  $p$  is called the *selection policy*.

The semantics are as follows:

- the network  $S_{\langle c \rangle p}^*(N)$  maintains an ordered *processing set* of indexed replicas of  $N$  over time;
- whenever it receives a record tagged by  $\langle c \rangle$ :
  - if the tag’s scalar value is strictly positive, it forwards the record to the replica indexed by the tag’s scalar value, creating the replica if necessary;
  - if the scalar is negative, it forwards the record to the replica indexed by the absolute value, then signals termination to the replica;
- if  $\langle c \rangle$ ’s value is zero, and for all records of another type, it routes the record to a replica according to the policy (described below). Moreover, if  $\langle c \rangle$ ’s value is zero, then the value of  $\langle c \rangle$  is automatically set upon entry by  $S^*$  to the index of the selected replica.

If there are no replicas currently defined upon receiving a  $\langle c \rangle$  record where  $\langle c \rangle$ ’s value is zero or upon receiving a record of another type, or if  $\langle c \rangle$ ’s value is negative and there is currently no replica indexed by the absolute value, the record is forwarded directly to the output stream.

The following policies are available:

- $S_e^*$  for *even* distribution: best effort is made to distribute the records evenly to the current replicas;
- $S_r^*$  for *last replica*: records are distributed to the last replica selected by  $\langle c \rangle$ ;
- $S_{l_a, h_a}^*$  for *lowest or highest available replica*: records are distributed to any replica currently able to accept input, preferring replicas with the lowest index or highest index respectively.

If the policy is not specified, it defaults to the even distribution.

Activity and termination are further determined for  $S^*$  like for selection and replication composition earlier.



### 4.3 Identifiers for run-time activities

As discussed in section 2 and more specifically in section 2.3, any static network specification translates, at run-time, into zero or more replicas, and for each replica, into a lifecycles of zero or more activations before eventual termination. Any run-time activity, either communication or processing or records, can thus be traced back to the static specification using:

- for transformations, the path to the sub-network defining the transformation, augmented at each level of nesting with an identifier for the replica where the transformation takes place and an identifier for the activation of that replica;
- for records, the identity of the transformation that has produced the record, augmented with the causal index<sup>7</sup> of that record.

To identify run-time activities, S<sup>+</sup>NET standardizes the notion of *network index*. A network index is a list of triplets  $(x, y, z)$  where  $x$  is the functional path through the specification,  $y$  an identifier for the replica and  $z$  an identifier for a particular activation. Functional paths and replica identifiers are defined “naturally” for functional combinators:

- for simple composition C and selection S, the functional path designates the position of the sub-network in specification order, and the replica identifier is typically 0 (although that may be changed by extra-functional combinators below);
- for ordered replicated composition C\*, the functional path is the position of the replica in the dynamic unfolding<sup>8</sup>, and the replica identifier is merely unique for that position;
- for unordered replicated composition C<sup>!</sup>, the functional path is an identifier for the processing subsequence on the input, the replica identifier is merely unique for that subsequence;
- for replicated selection S\*, the functional path is the scalar value that identifies which alternative to use, and the replica identifier is typically equal to the static path.

For example, consider the network

$$C(N, S(C^* \circ C(M, O), P))$$

or “N . . ((M . . O) \* |P)” in source form. An activity on behalf of the 3rd activation of the 12th replica of network  $N$  would be identified by  $[(0, 11, 2)]$ . The value 0 indicates the first argument of the first C combinator. Likewise, the index  $[(1, 0, 12); (0, 0, 12); (12, 0, 12); (0, 0, 12)]$  identifies an activity on behalf of the second position of the outer C, of the first position of S, of the 13th unfolding of C\* and at the first position of the inner C, i.e. in  $M$ .

From network indices we derive the notion *functional network indices*, which are lists formed by taking the first index of each element in a full network index. This notion is equivalent to the network indices defined in [6, App. B.3]. Functional network indices are often sufficient to establish functional causality for observed values, but they may lose information about extra-functional causes.

<sup>7</sup>The causal index is due to multiplicity: any input record processed by a box can cause zero, one or more output records, and these can need to be distinguished by other means than their type.

<sup>8</sup>The position is given prior to garbage collection, i.e. the index may be larger than the dynamic depth.

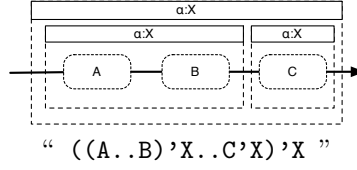


Figure 4: Graphical representation of  $\alpha_X \circ C(\alpha_X \circ C(A, B), \alpha_X \circ C)$

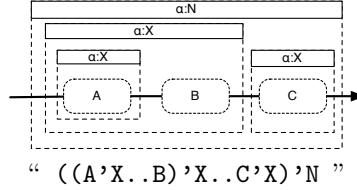


Figure 5: Graphical representation of  $\alpha_N \circ C(\alpha_X \circ C(\alpha_X \circ A, B), \alpha_X \circ C)$

#### 4.4 Network labeling and selection

The  $\alpha$  combinator helps naming and identifying run-time instances of subnetworks. The construct  $\alpha_X(N)$  (“N’X” in source) is a network specification if  $N$  is a network specification.  $X$  is called the *network label*. The functional semantics of  $\alpha_X(N)$  are those of  $N$ .

Network labels are used as anchors by the extra-functional features defined hereafter. There are two possible uses of anchors:

- the constructs  $\beta$ ,  $\theta$ ,  $\rho$ ,  $\gamma$ ,  $\tau$  and  $\phi$  refer to the set of *outermost inner network(s)* with some label. For example in fig. 5,  $N$ ’s outermost inner networks labeled by  $X$  are  $C(A, B)$  and  $C$ ;
- the constructs  $\delta$  and  $\phi$  refer to the *innermost outer network* using some label. For example in fig. 4,  $A$ ’s innermost outer network labeled by  $X$  is  $C(A, B)$ , whereas  $C$ ’s is itself.

Note that S<sup>+</sup>NET’s construct “**net**  $X \dots$ ” in source notation also implicitly generates a use of  $\alpha_X$  around the enclosed network specification in S<sup>+</sup>NET.

#### 4.5 Environmental exception handling

The network combinator  $\beta$  provides a facility to handle extra-functional exceptions in the run-time environment. The construct  $\beta_{X(a=E)}(N)$  (“N\$X(a=E)” in source) is a network specification if  $N$  is a network specification.  $E$  is called the *exception specification*;  $a$  is called the *exception label*.  $X$ , if specified, is called the *target network selector*.

$\beta$  applies to all the outermost inner networks labelled by  $X$ , or to the inner network directly if  $X$  is omitted. The selected network(s) are called *target networks*. Semantically, the input of  $\beta_{(a=E)}(N)$  is provided to  $N$  in-order, unchanged. If the processing of an input record  $r$  by any of the target networks causes an uncaught extra-functional exception of type  $E$ , it is reported as follows:

- all outputs of  $N$  produced causally from  $r$  are not output from  $\beta(N)$ ;
- if  $r$  has no field labelled  $a$ , or if  $a$ ’s value is non-zero, then the exception is propagated to the enclosing network; otherwise:

- the state of  $N$  is restored to prior to  $r$ 's input; and
- $r$  is re-injected as input to  $N$  with its scalar field  $a$  modified to a non-zero value (indicating an exception has occurred).
- whenever an exception escapes the scope of a  $\beta$  network, all the replicas of the inner network are terminated, even if they are active. Subsequent input records cause a new instantiation.

A  $\beta$  network may let subsequent records of the input flow in the protected network before the faulty record is retried without violating causality. This is possible to implement if the protected network is stateless, if the environment supports transactions and an aborted transaction has no impact on the processing of subsequent records, or if the specific application tolerates such reordering. To obtain determinism, that is, preservation of input order,  $R \circ \beta$  can be used.

Any exceptions generated by sub-networks others than the target network(s) are not caught and propagated to the enclosing environment instead.

## 4.6 Extra-functional isolation

The network combinators  $\theta$  and  $\theta'$  express extra-functional isolation requirements on the execution environment. The constructs  $\theta_{X/R}(N)$  and  $\theta_{X/R}^+(N)$  (“N/X/R” and “N/X/+R” in source) are network specifications if  $N$  is a network specification.  $R$  is called the *isolation property*;  $X$ , if specified, forms the target network selector. Both  $\theta(N)$ 's and  $\theta^+(N)$ 's functional semantics are those of  $N$ . Like  $\gamma$  and  $\tau$ ,  $\theta$  and  $\theta^+$  apply to the outermost inner networks labelled by  $X$ , or to the inner network directly if  $X$  is omitted; the selection is also called target network(s). The functional semantics of  $\theta(N)$  and  $\theta^+(N)$  are those of  $N$ .

Extra-functionally,  $\theta_R$  specifies that the execution environment guarantees that all replicas of the target networks are isolated *from each other* relative to property  $R$ . In contrast,  $\theta_R^+$  specifies that the target replicas are isolated *from each other and also from their enclosing network* relative to  $R$ . In particular with  $\theta$ , the management activities of the enclosing network need not be isolated from the replicas of the target networks; only  $\theta^+$  guarantees this isolation.

S-NET proposes the following isolation specifications:

- $\theta_f$  for *relative progress independence* (fairness), i.e. the progress of each replica not starved on input or blocked on output is guaranteed independently from other replicas (and from the enclosing network with  $\theta^+$ ). This may be implemented using preemptive time sharing;
- $\theta_b$  for *relative bandwidth independence*, i.e. the internal bandwidth of processors and channels onto which each replica is mapped is reserved and free of contention from other replicas (and from the enclosing network with  $\theta^+$ ). This may be implemented using round-robin real-time scheduling on one processor/channel, or via true hardware parallelism;
- $\theta_s$  for *relative storage independence*, i.e. the storage allocated by the running components and network management is sourced from storage pools separate between replicas (and from the enclosing network with  $\theta^+$ );
- $\theta_p$  for *relative energy supply independence*, i.e. the power demands of each replica are satisfied independently from the power usage of other replicas (and from the enclosing network with  $\theta^+$ ).

An implementation of S<sup>+</sup>NET may be unable to satisfy a  $\theta$  requirement at run-time. In this case, best effort is made to report this inability statically; oth-

erwise an exception of type  $\text{Violation}(R)$  is raised at the first network activation without the required guarantee.

## 4.7 Extra-functional budget

The network combinator  $\rho$  expresses extra-functional budget requirements on the execution environment. The construct  $\rho_{X:R}(N)$  (“ $\text{N/X:R}$ ” in source) is a network specification if  $N$  is a network specification.  $R$  is called the *budget specification*;  $X$ , if specified, forms the target network selector.  $\rho(N)$ ’s functional semantics are those of  $N$ . Like  $\gamma$ ,  $\tau$  and  $\theta$  previously,  $\rho$  applies to the outermost inner networks labelled by  $X$ , or to the inner network directly if  $X$  is omitted; the selection is also called target network(s). The functional semantics of  $\rho(N)$  are those of  $N$ .

Extra-functionally,  $\rho_{X:R}(N)$  specifies that the execution environment caps the extra-functional budget  $R$  available to all replicas of the target networks, to a proportion of the budget available to the surrounding network. The following budget specifications are supported:

- $\text{mp}(x)$  for *maximum power*, i.e. the amount of power collectively consumed by all target networks;
- $\text{mc}(x)$  for *maximum storage*, i.e. the amount of storage collectively allocated;
- $\text{mfl}(x)$  and  $\text{mll}(l)$  for *maximum first/last latency*, i.e. the maximum duration between the moment a record is input by a replica and the first/last output causally produced;
- $\text{mti}(x)$  and  $\text{mto}(x)$  for *maximum input/output throughput*, respectively;
- $\text{mdla}(x)$ ,  $\text{mdaa}(x)$  and  $\text{mdpa}(x)$  for *maximum dynamic liveness/activity/agent arity*, respectively.

The parameter  $x$  establishes a budget relative to the budget available to the enclosing network. This can be either a ratio (e.g. 10%) or maximum absolute value (e.g. 10ms). The coordination layer makes a best effort to enforce the requirement, possibly throttling the execution strategy and processing rates to match the constraint. If a maximum value cannot be satisfied from the outer budget, or when a network is known to behave in violation with the requirement, an exception of type  $\text{Violation}(R)$  is raised at run-time. This may happen while a box is computing; for example, with  $\rho_{\text{mfl}}$  if a latency bound is exceeded while a box is still transforming its input record, the behavior can be aborted preemptively.

## 4.8 Projections: mapping specifications into processing agents

The network combinators  $\gamma$  and  $\tau$  express how run-time agents are spawned to process activations.  $\gamma$  chooses between entity-centric and record-centric projections, and thus helps control locality and the trade-off between computation and communication (cf. section 5.6).  $\tau$  decides the lifetime of agents, and thus helps control the trade-off between jitter and throughput (cf. section 5.7).

The constructs  $\gamma_X^e(N)$ ,  $\gamma_X^r(N)$ ,  $\tau_X^\omega(N)$  and  $\tau_X^\epsilon(N)$  are network specifications if  $N$  is a network specification. They are noted  $\text{N/X!ge}$ ,  $\text{N/X!gr}$ ,  $\text{N/X!to}$ ,  $\text{N/X!te}$  in source form.  $X$ , if specified, is the target network selector. Their functional semantics are those of  $N$ . As with the previous combinators, their semantics

apply to the target network(s) selected by  $X$ , or to the inner network directly if  $X$  is omitted.

- $\gamma^e$  specifies that the target networks should be mapped to run-time agents using an *entity-centric projection*, that is, agents are created for the entities in the specification. For example, for the network  $C(A, B)$  one agent is created for  $A$  and another for  $B$ . Agents communicate over buffered channels, which implement the S<sup>+</sup>NET streams, and terminate when the corresponding functional entity terminates as per section 2.3;
- $\gamma^r$  specifies that the target networks should be mapped to run-time agents using a *record-centric projection*, that is, agents are created for each successive input records. For example, for  $C(A, B)$  one agent for a first input record  $r_1$  executes code for  $A(r_1)$  then code for  $B(A(r_1))$ ; another agent for  $r_2$  executes code for  $A(r_2)$  then code for  $B(A(r_2))$ ;
- $\tau^\omega$  specifies that agents created for the target networks should be *lingering*: once an agent is created, it tries to consume records/entities as long as it can. Agents with  $\gamma^r$  may terminate at transducers and reordering points, at the latest at the output edge of  $N$ . Agents with  $\gamma^e$  terminate when a network terminates;
- $\tau^\epsilon$  specifies that agents for the target networks should be *ephemeral*: an agent only runs for the duration of an “elementary” processing: with  $\gamma^e$ , only for one agent cycle (cf. section 2.3); with  $\gamma^r$ , only for one step through the network graph.

Agents correspond to cooperatively scheduled tasks, threads, processes or virtual machines depending on the implementation and the requirements expressed via  $\theta$  and  $\rho$  (cf. sections 4.6 and 4.7). Agents are identified by the full network index of the activation that created them. Agents may run concurrently, either interleaved over time on sequential processors or simultaneously on parallel hardware. With  $\gamma^e$  the concurrency of agents stems from the concurrency between the logical entities in the specification; whereas with  $\gamma^r$  the agent concurrency stems from stream concurrency.

From the projection of specifications into agents we derive the notion of *dynamic agent arity* for a sub-network, which is the current number of running agents for this sub-network.

#### 4.8.1 Composability of $\gamma^r$ and $\gamma^e$

Any  $\gamma^r$  sub-network maps to a set of agents under control of a single “input” agent which reads the input records of the entire sub-network from its surrounding environment. Moreover, all the agents of a  $\gamma^r$  network *eventually* synchronize so that all final outputs of the network appear as if produced by a single “output” agent. Therefore, any  $\gamma^r$  network is a valid operand for combinators captured within a  $\gamma^e$  combinator. For example, the network

$$\begin{aligned} & \gamma^e \circ S (\gamma^r \circ C(M, N), \gamma^r \circ C(O, P)) \\ & \text{“ ( (M..N)/!gr | (O..P)/!gr ) /!ge ”} \\ & \text{(also: “ ( (M..N)'x | (O..P)'x ) /!ge /x!gr ”)} \end{aligned}$$

expresses that the transformations for  $C(M, N)$  and  $C(O, P)$  should be carried out in a record-centric fashion, but that different (groups of) entity-centric agents should be used to create a pipeline-like parallelism between the subnetworks of the selection.

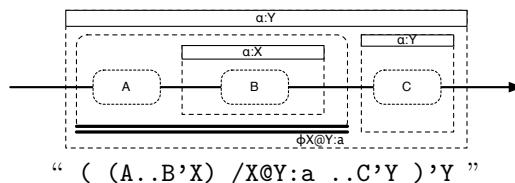


Figure 6: Graphical representation of  $\alpha_Y \circ C(\phi_{X,Y:a} \circ C(A, \alpha_X \circ B), \alpha_Y \circ C)$

Conversely, all agents created for a  $\gamma^e$  network can be drained of input, and their left-over state serialized to be re-instantiated later or elsewhere. Therefore, any  $\gamma^e$  network is also a valid operand for a  $\gamma^f$  network.

#### 4.8.2 Sequential execution

$\rho_{\text{mdpa}}$  can interact with  $\gamma$  to obtain sequential execution. For example,  $\gamma^f \circ \rho_{\text{mdpa}(1)}$  forces each input record to be processed, in turn, sequentially through the inner network.  $\gamma^e \circ \rho_{\text{mdpa}(1)}$  forces the first entity to process the entire input stream sequentially, accumulating all its output records in temporary buffers, only then lets the second entity process all the intermediate records sequentially, and so forth.

### 4.9 Hardware affinity and mapping

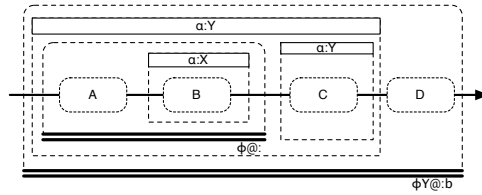
Resource mapping assumes that hardware resources are structured in a tree that reflects locality and granularity, and where the leaves are processing and storage units; for example cluster nodes at the top level, then processor sockets, then processor cores, with hardware threads and memories as leaves. The tree need not be homogeneous nor balanced. Each node in the tree is *enumerable* (by index from its parent node) and some nodes may also be annotated by *metadata* that indicate special features, e.g. hardware accelerators or specialized (IP) cores.

There are two parts to resource mapping in  $S^+NET$ : *assignment* and *placement*. Assignment establishes a mapping from a set of sub-networks sharing a common network label to a node in the hardware resource tree, possibly higher than leaves. Placement occurs at the point processing agents are created, using the resources previously assigned to the corresponding sub-network. Once placed, an agent stays placed at the same resources until it terminates; the  $\tau$  introduced earlier in section 4.8 thus helps control opportunities for work migration.

The combinator  $\phi$  only helps manage assignment of hardware resources. Placement is automatically managed by the coordination layer to satisfy the demands of  $\theta$  and  $\rho$  within the resources assigned by  $\phi$ . However,  $\delta(h)$  (cf. section 4.10) can observe placement after it is computed.

#### Semantics in $S^+NET$

The construct  $\phi_{X,Y:a}(N)$  (“ $N/X@Y:a$ ” in source) is a network specification if  $N$  is a network specification.  $a$  is called the *assignment specification*;  $X$  is the target network selector and  $Y$  is called *origin selector*. The functional semantics of  $\phi_{X,Y:a:p}(N)$  are those of  $N$ .



“ ( ( A . B ' X ) / @ . . C ' Y ) ' Y . . D ) / Y @ : b ”

Figure 7: Graphical representation of  $\phi_{Y:b} \circ C(\alpha_Y \circ C(\phi \circ C(A, \alpha_X \circ B), \alpha_Y \circ C), D)$

Like the previous combinators,  $\phi$  applies to the target network(s) selected by  $X$  or the inner network directly if  $X$  is omitted. It also refers to the innermost outer network labelled by  $Y$ , or to the innermost outer network targeted by an outer  $\phi$  if  $Y$  is omitted; this selection is called *origin network*. For example in fig. 6,  $\phi$ 's target network is  $B$  whereas its origin network is the entire group  $C(A, B, C)$ . In fig. 7, the inner  $\phi$ 's target network is  $C(A, B)$  because there is no target selector. Its origin network is the one labeled by the outer  $Y$ , i.e. the entire group  $C(A, B, C)$ , because that is the innermost outer network targeted by the outer  $\phi$ .

Extra-functionally,  $\phi$  assigns sub-resources from the origin network to replicas of the target networks. If  $a$  is omitted, the construct inherits the resources assigned to the origin network. For example in fig. 7, the resources assigned are inherited from the outer  $Y$  (therefore this inner  $\phi$  does not specify anything useful).

The following assignment specifications are supported:

- $\text{share}(x)$ : all replicas of the target networks are commonly assigned the sub-resources selected by  $x$ ;
- $\text{split}(x)$ : each new replica is assigned one of the sub-resources selected by  $x$ .

The syntax of  $x$  will be detailed in future work; it permits either to select a path in the resource tree (to achieve resource partitioning) or filter sub-trees based on a predicate on metadata (to select specialized resources), relative to the resources assigned at the origin network. If there are no sub-resources selected, or if new replicas run out of unassigned sub-resources, an exception of type `Exhaustion` is raised.  $\delta(h)$  can be used to inspect the arity of the selected sub-resources and the number of sub-resources yet unassigned.

#### 4.10 Environmental awareness

Environment observations are noted  $\delta_{\langle t \rangle, a}(E)$  in algebraic form or “[<t>.a=E]” in source form.  $\langle t \rangle$  is called the *matching tag*;  $a$  is called the *payload field*, and  $E$  is the *environment function*.

A  $\delta$  network reproduces its input records to its output stream unchanged and in-order, except that for each record consumed that matches tag  $\langle t \rangle$ , the value of the field  $a$ , if present, is updated depending on  $E$  before forwarding the record. If  $\langle t \rangle$  is not specified,  $a$  is modified in all records that contain the field label, regardless of tag. If  $a$  is not specified, the scalar value of the tag itself is modified. Implementations of S<sup>+</sup>NET provide at least the following environment functions:

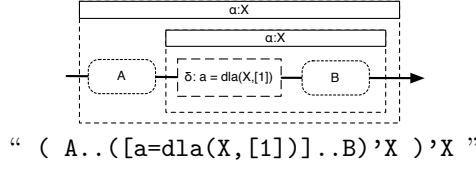


Figure 8: Graphical representation of  $\alpha_X \circ C(A, \alpha_X \circ C(\delta_a(\text{dla}(X, [1])), B))$

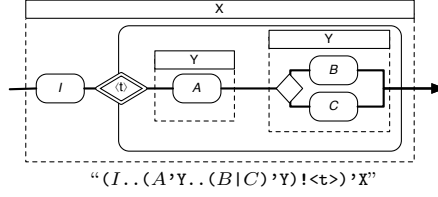


Figure 9: Graphical representation of  $\alpha_X \circ C(I, S_{(t)}^* \circ C(\alpha_Y \circ A, \alpha_Y \circ S(B, C)))$

- $\text{time}(g)$ : difference between the current time and the scalar's previous value;
- $c(X, i, g)$ : current amount of storage used by activities;
- $p(X, i, g)$ : current power used by activities;
- $\text{ti}(X, i, g)$ ,  $\text{to}(X, i, g)$ : current input/output throughput;
- $\text{fl}(X, i, g)$ ,  $\text{ll}(X, i, g)$ : current first/last output latency (from the input of a record  $r$  and the first/last output causally produced from  $r$ );
- $\text{dla}(X, i)$ ,  $\text{daa}(X, i)$  and  $\text{dpa}(X, i)$ : current dynamic liveness/activity/agent arity, respectively;
- $\text{h}(X, i, p)$ : value of the property  $p$  of the hardware resources currently assigned.

For each function parameterized by a network label  $X$  and functional index  $i$ , the observation pertains to the sub-network identified by  $i$  relative to the innermost outer network labeled by  $X$ . For example in fig. 8, the observation pertains to network  $B$ . The network index is necessary to disambiguate which network to observe when the same label is used multiple times; future work will explore semantics to gather observations across multiple sub-networks.

Throughputs are expressed in records per  $g^{-1}$  seconds, power in  $g^{-1}$  watts, time and latencies in multiples of  $g^{-1}$  seconds, and storage in multiples of  $g$  bytes. The values are averaged over time using a sliding window or exponential smoothing. If the environment does not support a function  $E$  at run-time, an exception of type `Unimplemented` is raised.

#### 4.11 Implementation services

Next to the application specifications constructs defined so far, the S<sup>+</sup>NET execution environment provides the following services, available to the operator of a running application:

- *lookup*, which given a run-time event and a network label, produces the network index *relative* to the most inner  $\alpha$  construct that identifies the run-time event.

To illustrate, consider for example the application in fig. 9. Consider



then a run-time observation of an activity related to the inner  $B$  network instantiated from tag  $\langle t \rangle = 123$  in the replication selection. When looked up from label  $X$ , the functional network index is  $[1; 123; 1; 0]$ ; when looked up from label  $Y$ , the network index is simply  $[0]$ .

- *arity inspection*, which given a network index approximates the activity/liveness/agent arity of the replica identified by  $n$ .

To illustrate, consider the example from fig. 9 after  $\langle t \rangle$  has run through the network with values 123 and 124. In a single instance of  $X$ , the liveness arity for both sub-networks identified by  $Y$  is 2.  $A$  has two live replicas  $[1; 123; 0]$  and  $[1; 124; 0]$ ; whereas  $S(B, C)$  has replicas  $[1; 123; 1]$ ,  $[1; 124; 1]$ .

- *state inspection*, which given a network index enumerates all the inner constructs that currently have state and the type of state they hold. Active transducers are listed with the name of the state they are currently in and the content of their currently full hold variables; active reorderings (R) with the number of records currently held; replications with the number of replicas currently maintained, etc.

The lookup service is akin to the reverse lookup service commonly found in debuggers. The arity and state inspection services are intended to provide a handle on the current management state maintained by the coordination layer.

## 5 Relationship to S-NET and design rationales

### 5.1 Overview of functional changes to S-NET

$S^+$ NET uses the same streaming foundations as S-NET and reuses its primitive networks for boxes and filters. There are however three major functional updates in  $S^+$ NET, motivated by practical experiences using S-NET. As we discuss in section 5.2,  $S^+$ NET introduces exact type match and dedicated channels via type tags where S-NET uses mostly structural record subtyping. As we motivate in section 5.3,  $S^+$ NET merges S-NET’s filters and synchrocells into the single finite state machine construct called transducer.  $S^+$ NET furthermore introduces unordered replication composition, as motivated in section 5.4.

Next to these major updates, a few aspects of S-NET have been clarified and simplified in  $S^+$ NET.  $S^+$ NET makes reordering orthogonal to other functional composites via its reordering combinator, while S-NET provides separate variants of combinators that preserve input order. While S-NET solves the routing ambiguity in its selection combinator “non-deterministically” without stating what strategy is actually used,  $S^+$ NET specifies it; any non-determinism, if so desired, can be expressed *and controlled* using extra-functional selection (cf. section 4.2).

### 5.2 Stream connections and structural typing

Both S-NET and  $S^+$ NET are based on the transformation of a general acyclic application graph into a serial-parallel structure with bypasses for non-matching types. Support for cyclicity was added in S-NET via replication composition. The overall design seems to work very well with most of our industrial use cases. It is not free, however, from certain design drawbacks.

First of all, the serial-parallel representation, i.e. a pipeline of groups of boxes

connected in parallel, generally requires a bypassing mechanism as the original transformation suggested. To understand this requirement, one can think of a node of the graph sending directly to another node, whose maximum distance from the sources of the graph is greater by more than one than that of the sending node. After the transformation those nodes will be separated by one or more pipeline stages that would have to be bypassed. The original S-NET design did not appreciate the ubiquity of bypasses in any real program, even though special compact notation ( $\rightarrow$ ) was included to avoid long-handed specifications.

From the bulk of experience that we have accumulated over the recent years it does appear that the remedy should come from the type system. Indeed, already (and according to some, rather inelegantly) tags in S-NET are classified into ordinary and binding, the latter category intended for pruning the flow inheritance tree that results from combining input types when the most general acceptable set of records is being determined. Binding tags are indeed not only necessary but they should be seen as “normal” as they are the only ones that can separate variants. For example a network accepting  $\{a\}$ ,  $\{b\}$  and  $\{c\}$  as three variants and one which has a response to each of those would indeed have to have a response to all seven distinct nonempty unions of these sets as various subtypes of the originals. Worse still, all but the singleton sets of these unions would lead to a nondeterministic choice in identifying which original variant they should be assigned to, with the extra members of the set being flow-inherited. Unfortunately, explicitly separating the variants with tags in the spirit of algebraic data types as well as standard practice of imperative programming languages does not solve the problem: the variants  $\{\langle i \rangle, a\}$ ,  $\{\langle ii \rangle, b\}$  and  $\{\langle iii \rangle, c\}$  are as prone to the undesired inheritance as the original ones because the tags, too, can be inherited unless they are binding. Consequently in this example only  $\{\langle \#i \rangle, a\}$ ,  $\{\langle \#ii \rangle, b\}$  and  $\{\langle \#iii \rangle, c\}$  represent a solution that is free of spurious subtyping, as it only accepts records that contain either a or b or c as befits a proper variant record type.

In the light of this, we found desirable to simplify the typing rules of S-NET and promote a single form of tags that is always binding, and explicitly defines the end-points of streams. Such tags also inhibit inheritance but in a different way: while S-NET’s binding tags demand that the receiving end of the match offers the same tag or else this would be a type error, when S<sup>+</sup>NET’s new tags fail to match they cause the record they belong to to be bypassed over the unmatched entity. S<sup>+</sup>NET’s tags delay a type error, rather than merely causing it to be ignored. If a record carrying a tag reaches the exit of a net environment, or fails to match an explicit network type signature, a type error does result.

### 5.3 Usability of synchrocells

Another problem was that S-NET does not have a satisfactory solution for is the variety of synchronisers that are required in the real world. Its simple  $n$ -ary synchrocells have a finite and rather trivial state machine behaviour. The idea behind them was originally that any behaviour more complex than that would be realised via unfolding replicative structures that contain synchrocells and boxes that compute transitions between various states of the required compound synchroniser. For instance if the application needs a synchrostructure that combines records of type 1 with some persistent record value of type 2, which is also periodically updated by records of type 2, such a structure could

be realised as a network where records of type 2 cause synchronisation in a synchro-queue while at the same time duplicating themselves to be sent to the next stage of the queue thus mimicking real persistency. Unfortunately, this method is not just awkward, it also requires a really inelegant action when the persistent values are to be modified. At such moments, a spurious record of type 1 would need to be produced only to be discarded at the exit of the network (for which a tag should be flow-inherited to indicate that the record is spurious). These regrettable manoeuvres seem unavoidable in the original vision while at the same time they are also illogical as far as the basic principle is concerned. The principle was to divide the boxes into “can compute, have no state” and “have a state, cannot compute”, the latter in the sense of modifying box values. This principle does not exclude synchronisers with complex behaviour as long as all they do is select what to store and what to pass on. The original S-NET synchrocells seem as undeservedly specialised for one kind of synchronisation task as would boxes with only one output variant be for one task of computing. To make matters worse, the behaviour described above is very plausible: any application that wishes to tune the parameters of its algorithm from time to time would need the above kind of synchroniser as a building block.

The solution lies in introducing a synchro-construction language similar to the filter language from S-NET. Such a language can be simple and elegant, allowing the coordination programmer to specify state transitions between the synchroniser states as well as output and stored values in terms of set operations on input and stored records. The state transitions should logically be allowed to depend on values found in input records. It is easy to satisfy oneself that the corresponding transducer language in S<sup>+</sup>NET is not much larger than the filter and synchrocell languages from S-NET. Besides, as soon as this language exists, it becomes possible to express stateless synchronizers that perform simple tasks on input records, and the filter construct from S-NET can be dropped in favor of the more general construct.

## 5.4 Aggregate updates

The data flow model on which S-NET is built requires complete encapsulation: all that a box can read is its input record and all that it is supposed to write into is output records. This has worked surprisingly well in most use cases. However, when the application requires parallel processing of a large shared datastructure, such as a large graph or a distributed database, the basic assumption of dataflow could easily become burdensome. What is required is a slightly different discipline: sharing should be allowed but the box must yield to the coordinator’s control when it attempts sharing. Specifically, we anticipate the need to access a large shared data structure from a box and only to read and modify a small part of it; we expect that such modifications done by different boxes collide infrequently and that when they do a transaction discipline must be enforced to preserve serialisability as its essential semantics. In other words a box should be able to access the shared object, read a small part of it and modify a small part of it and then terminate (as boxes usually do), without any other box modifying the data being read by the first box before it terminates. The ordering of such modifications would be linear but nondeterministic, which in practice means optimistic parallelism with failures and retractions.

In terms of S-NET language constructs, we are lacking a combinator which

would connect replicas of its operand serially in non-deterministic order and stream the data structure through them, while at the same time delivering parameter records to the replicas in a parallel fashion: a non-deterministic parallel-serial combinator. This combinator is introduced as unordered replication composition in S<sup>+</sup>NET.

## 5.5 Identification of run-time activities

Perhaps the most painful shortcoming of S-NET while using it in practice is the lack of insight during application execution to relate a run-time behavior back to its origin in the application specification.

For example, a common situation is to detect that a particular procedure in the S-NET application’s binary code concentrates most of the execution time, but only for some of its activations. This situation can appear when a component box has different behavior classes depending on the data provided as input. The question then appears: in which path through the application network is this behavior class triggered?

A textual locus in the static application specification is usually not sufficient to isolate the origin of a behavior. With replication, a single component may be instantiated multiple times during execution, with all replicas executed concurrently. In general, the run-time identifier of a network replica is necessary to identify the cause of an observation. Another realization is that not only component code is worthy of the optimization engineer’s attention. In [7], the authors demonstrate that internal S-NET management tasks were the cause of a load imbalance in a parallel computation. Unfortunately, in contrast to component boxes all the internal tasks of S-NET are anonymous and cannot be easily related to the input application specification.

What these scenarios have revealed is that S<sup>+</sup>NET, like any programming environment, needs a “reverse lookup” mechanism from space/time loci during execution back to a specification’s source code. However, a simple mechanism based on a static mapping of memory addresses to source locations, like those used by traditional debuggers, would be insufficient for a highly concurrent environment where multiple processes running at the same address may emerge from multiple source locations or paths through the specification. Consequently, we introduce a naming scheme with S<sup>+</sup>NET (section 4.3), a new labeling combinator  $\alpha$  and selection scheme (section 4.4), and run-time services to inspect a running application (section 4.11).

## 5.6 Entity-centric vs. record-centric projections

When S-NET was originally designed, its language operators were constructed as abstractions of common patterns when engineering applications made of concurrent processes. The unphrased assumption behind the design of S-NET was that entities in a specification should abstract run-time processes, and conversely that each entity would be reified during execution using a process; and thys that inter-component streams would be naturally implemented as buffered channels connecting the processes together. In this vision, a run-time execution of an S-NET application defines processes for each box, plus additional control processes for the composition operators: “splitter” processes before parallel composition to redirect records to the sub-processes depending on their type,

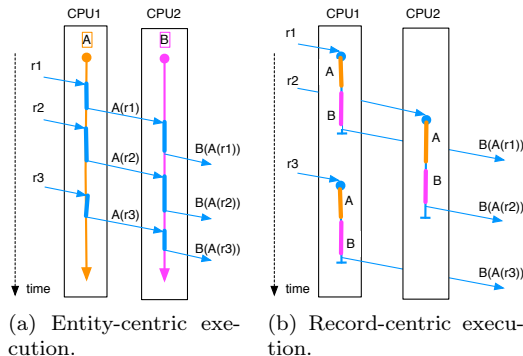


Figure 10: Two valid run-time projections of  $A . B$  onto processes.

“merger” processes after composition to restore order, “synchronizer” processes for synchronocells, etc.

Meanwhile, the functional definition of S-NET did not altogether mandate that an implementation must use this *projection* onto process networks to actually carry out the application’s execution. This under-specification led multiple researchers to realize later on, separately, that there exist other possible run-time projections of an S-NET specification.

One such approach can be found in [6], where the author suggests to create processes for each individual *record*, instead of box or control entity. Each record-process then computes sequentially and recursively the transformation operated for that input record over the length of the S-NET application specification. To illustrate this approach, we can consider for example the pipeline of  $A$  and  $B$  in sequence, i.e.  $A . B$ . The “natural” projection of this network onto processes is given in fig. 10a: one process is created to compute transformations of inputs by function  $A$ ; another process is created to compute transformations by  $B$ . The “transposed,” record-centric projection is given in fig. 10b: one process is created for each input; each process sequentially computes  $A$  then  $B$  for this input, then terminates. It is easy to see that both projections are *functionally equivalent*, in that they compute the same relationship between input and output according to the S-NET semantics.

In a record-centric vision, stream synchronization can be implemented by joining concurrent threads; boxes with multiplicity by creating new streams for any new additional record injected in the network. The reader is referred to [6, Chap. 7] for a detailed description of the mechanisms involved. The existence of multiple valid projections of S-NET was quickly recognized as more than a mere intellectual curiosity. Indeed, the projection that maximizes throughput or reduces latency for a given execution platform *depends on the platform’s parameters* and the algorithmic complexity of the boxes themselves.

An entity-centric projection is desirable for compute-bound applications, when the objective is to specialize processors to the component’s function in order to accelerate them. However, an entity-centric projection pays a price in locality: the intermediate results between components travel spatially and internal communication costs are increased. If the platform does not provide sufficient bandwidth between processors, this projection quickly becomes communication-

bound. In practice, we have seen this happen when the computational complexity of components was not enough to mask the limited memory bandwidth available to multiple cores in a shared-memory system, or when attempting to map S-NET applications to platforms with GPU accelerators.

Conversely, a record-centric projection is desirable for applications with large communication requirements on their internal streams compared to the requirement on their input and output streams, i.e. when the objective is to maximize locality. A record-centric projection pays a price in efficiency. Each processor must be sufficiently general to perform all the transformations in the application, and code locality can become an issue. Moreover, synchronization state is likely to be shared across arbitrary processors in the system, and synchronization operations may thus have higher latencies.

The existence of this duality and its consequences w.r.t optimization opportunities suggests to equip S<sup>+</sup>NET with mechanisms to let an optimization engineer select which type of execution projection to use, separately from the functional specification of the application. This is introduced via the  $\gamma$  combinator (cf. section 4.8).

## 5.7 Lifetime of activities

The most recent entity-centric implementation of S-NET, currently used as research vehicle, uses cooperative scheduling of lightweight tasks over worker threads pinned to hardware processors [11]. Cooperative scheduling lowers the cost of concurrency management overall by avoiding context switches to the operating system and maintaining separate task state and scheduling queues per worker. However, cooperative scheduling, by construction, places the scheduling responsibility in the hands of the tasks themselves: once started by a worker scheduler, it stays assigned to that worker until it terminates. Extending the implementation to allow dynamic task migration while a box-task is transforming records is undesirable for two reasons: it would require to re-introduce the overhead of preemption, as well as mutual exclusion for task management state between workers, which the lightweight task/worker infrastructure was intended to avoid in the first place. In contrast, a mapping decision prior to a task’s creation is cheap to implement since there is no state yet to synchronize nor activity to preempt. Intuitively, an application’s execution where tasks are short-lived thus offers more opportunities for cheap dynamic load balancing than a system where tasks persist after their creation.

This is the starting point of a modified implementation of S-NET that was finalized in the second half of 2012: instead of projecting S-NET entities into long-lived tasks that repeatedly read from their input stream and process one input record, in a loop (fig. 10a), this implementation causes each entity-task to terminate as soon as one record is processed, so as to allow a new mapping decision to be made for the next input record (fig. 11).

In other words, by shortening the *lifetime* to a task to the minimal amount of computation, that is the handling of exactly one input record by a box, we obtain the maximum flexibility for scheduling short of re-introducing fully-fledged preemption. However, this comes at a cost: the number of tasks created and cleaned up by unit of time increases, which amounts to re-introducing some management overhead. For homogeneous workloads this overhead is as undesirable as it is unnecessary, since homogeneity implies that fine-grained scheduling

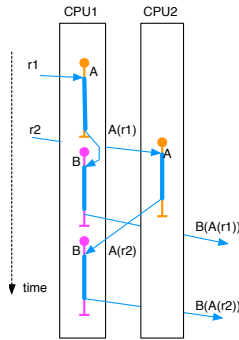


Figure 11: Short-lived entity-centric tasks for A..B.

decisions have a low impact on overall execution time.

While we have understood these trade-offs in the specific context of cooperative scheduling of tasks over worker threads, this context is merely an instance of a more general, fundamental aspect of concurrency management. Regardless of the mechanism used to implement concurrency over hardware processors, the chosen granularity for non-preempted, non-migrated activities determines a trade-off between throughput and jitter. Furthermore, there does not seem yet to exist any consensus about the best way to detect the optimal granularity for a given workload/platform combination. In particular, any approach which sets global values (e.g. the HZ property that sets the time slice duration on Unix systems, commonly set to 1 or 20ms) is typically inappropriate in the light of heterogeneous applications or resources where the granularity should change depending on the application part, possibly dynamically.

With S<sup>+</sup>NET, we thus propose to explore this trade-off using a novel approach based on two principles. First, we state that a computing agent (either a task, thread, or any other mechanism used to perform an elementary computation on the platform at hand) is assigned to hardware resources upon its creations and is not migrated throughout its lifetime. This assignment can be guided with the  $\phi$  combinator (cf. section 4.9). Then, we establish that each transformation specification in the S<sup>+</sup>NET abstraction can be projected not onto only one, but multiple successive agents during execution; this granularity can be in turn controlled via the  $\tau$  combinator (cf. section 4.8). This way, the throughput/jitter trade-off can be controlled flexibly for each sub-network in an application.

## 5.8 Environmental awareness

The practical application of S-NET has revealed consistently that some tuning parameters in an application specification are highly dependent on the execution resources actually available at run-time.

For example, a common pattern found in compute-bound applications is the divide-and-conquer network: a “splitter” box splits large input records into substreams of smaller records to be processed concurrently by parallel “compute” boxes. Users of this pattern typically face a challenge: how to decide the size of the smaller workloads?

In this example, there are two main scenarios. When the workload is heterogeneous, an incentive exists to split the workload in smaller sizes to give the coordination layer more opportunities to balance load dynamically across hardware resources. However, as the granularity becomes finer, the overheads of coordination increase relative to computation time; after a threshold the coordination overheads dominate and the throughput is actually reduced. Of course, this threshold depends both on the application *and the execution platform*: any specification-time granularity choice must be revised after profiling.

When the workload is homogeneous, an incentive exists to spread the workload evenly across the available hardware resources. The granularity should be as coarse as possible to minimize coordination overheads while obtaining the maximum parallelization possible on the platform. However, two obstacles prevent an optimal specification-time choice. For one, the size of memory caches in the execution platform determines a threshold past which cache thrashing will actually reduce throughput. Second, if more concurrent transformations are defined than there are processors available at run-time, context switch overheads also reduce throughput scalability.

This example is only one instance of a need for an *environmental feed-back loop*, whereby the coordination engineer can specify tuning parameters, such as sub-workload granularity selection in the example, as a function of the resources actually available on the execution platform. In S<sup>+</sup>NET, we propose to introduce such feed-back loops via the primitive network  $\delta$  (cf. section 4.10) which observes the execution environment and injects observations as scalar values that can be used in the application’s coordination logic. These can be then combined with  $\rho$ ,  $\theta$  and  $\phi$  to tune the application’s behavior according to extra-functional requirements and budgets.

## 6 Summary and conclusions

This report has presented the design of S<sup>+</sup>NET, a declarative language for describing networks of asynchronous components and their execution semantics in resource-constrained execution environments. While it reuses concepts from its predecessor S-NET, S<sup>+</sup>NET simplifies S-NET’s set of functional combinators and adds a more comprehensive synchronization facility, the transducer. In the extra-functional domain, S<sup>+</sup>NET provides facilities for:

- naming of networks and non-local name references to coordinate behavior;
- environment awareness, to react to resource availability;
- fault tolerance by enabling explicit exception handling;
- constraining the operational semantics of arbitrary sub-networks towards either a process-centric projection or a record-centric projection;
- constraining the mapping of networks to processing resources;
- constraining the execution to satisfy resource budgets.

The definition of S<sup>+</sup>NET is an effort concurrent to the definition of AstraKahn, another coordination language by the same research groups. The main difference between S<sup>+</sup>NET and AstraKahn is that all valid S<sup>+</sup>NET programs are serializable, whereas valid AstraKahn programs may contain non-serializable network cycles.



## Acknowledgements

This research was partly supported by the European Union under grant number FP7-248828 ADVANCE<sup>9</sup>. The authors would like to thank the ADVANCE partners for the constructive criticism of S-NET that has resulted in S<sup>+</sup>NET.

## References

- [1] C. Grelck. The essence of synchronisation in asynchronous data flow. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1159–1167, May 2011. ISSN 1530-2075. doi:10.1109/IPDPS.2011.277.
- [2] C. Grelck and F. Penczek. Implementation architecture and multithreaded runtime system of S-Net. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 60–79. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24451-3. doi:10.1007/978-3-642-24452-0\_4.
- [3] Clemens Grelck, Jukka Julku, and Frank Penczek. S-Net for multi-memory multicores. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 25–34. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-859-9. doi:10.1145/1708046.1708054.
- [4] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(1):221–237, 2008.
- [5] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010. doi:10.1007/s10766-009-0121-x.
- [6] Philip Kaj Ferdinand Hölzenspies. *On run-time exploitation of concurrency*. PhD thesis, University of Twente, Enschede, the Netherlands, April 2010. Available from: <http://doc.utwente.nl/70959/>, doi:10.3990/1.9789036530217.
- [7] Kenneth MacKenzie, Philip Kaj Ferdinand Hölzenspies, Kevin Hammond, Raimund Kirner, Nguyen Vu Tien Nga, Rene te Boekhorst, Clemens Grelck, Raphael Poss, and Merijn Verstraaten. Statistical performance analysis of an ant-colony optimisation application in S-Net. In Clemens Grelck, Kevin Hammond, and Sven-Bodo Scholz, editors, *Proc. 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures*, 2013. Available from: <http://www.project-advance.eu/wp-content/uploads/2012/07/proceedings.pdf>.
- [8] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Rémi Barrière, and Eric Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*,

---

<sup>9</sup><http://www.project-advance.eu/>

1(1):2079–2088, 2010. ISSN 1877-0509. ICCS 2010. Available from: <http://www.sciencedirect.com/science/article/B9865-506HM1Y-88/2/87fcf1cee7899f0eeaadc90bd0d56cd3>, doi: 10.1016/j.procs.2010.04.233.

- [9] Frank Penczek, Stephan Herhut, Sven-Bodo Scholz, Alex Shafarenko, Jung-Sook Yang, Chun-Yi Chen, Nader Bagherzadeh, and Clemens Grelck. Message driven programming with S-Net: methodology and performance. In *Proc. 3rd International Workshop on Programming Models and Systems Software for High-End Computing (P2S2'10), San Diego, USA*, 2010.
- [10] Frank Penczek, Jukka Julku, Haoxuan Cai, Philip Kaj Ferdinand Hölzenspies, Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. S-Net language report, version 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, AL10 9AB, United Kingdom, April 2010.
- [11] Daniel Prokesch. A light-weight parallel execution layer for shared memory stream processing. Master's thesis, TU Wien, Karlsplatz 13, A-1040 Wien, February 2010.
- [12] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, October 2009. ISBN 978-0521844253.
- [13] Alex Shafarenko. Non-deterministic coordination with S-Net. In Wolfgang Gentzsch, Lucio Grandinetti, and Gerhard Joubert, editors, *High Speed and Large Scale Scientific Computing*, number 18 in Advances in Parallel Computing. IOS Press, 2009. ISBN 978-1-60750-073-5. doi: 10.3233/978-1-60750-073-5-74.
- [14] M. Verstraaten. High-level programming of the Single-chip Cloud Computer with S-Net. Master's thesis, University of Amsterdam, Amsterdam, the Netherlands, January 2012.
- [15] Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, and Chris R. Jesshope. Mapping distributed S-Net on to the 48-core Intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium*. KIT Scientific Publishing, September 2011. ISBN 978-3-86644-717-2. Available from: <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937>.
- [16] Jeroen Voeten. On the fundamental limitations of transformational design. *ACM Trans. Des. Autom. Electron. Syst.*, 6(4):533–552, October 2001. ISSN 1084-4309. doi:10.1145/502175.502181.