

S-Net: A Declarative Approach towards Streaming Networks of Asynchronous Components

Clemens Grelck^{1,2}, Sven-Bodo Scholz¹, and Alex Shafarenko¹

¹ University of Hertfordshire
Department of Computer Science
Hatfield, United Kingdom
{c.grelck,s.scholz,a.shafarenko}@herts.ac.uk

² University of Lübeck
Institute of Software Technology and Programming Languages
Lübeck, Germany
grelck@isp.uni-luebeck.de

Abstract. We present the design of S-NET, a coordination language and component technology based on stream processing. S-NET achieves a near-complete separation between application code, written in a conventional programming language, and coordination code written in S-NET itself. S-NET boxes integrate existing sequential code as stream-processing components into streaming networks, whose construction is based on algebraic formulae built out of four network combinators. Subtyping on the level of boxes and networks and a tailor-made inheritance mechanism achieve flexible software reuse.

1 Introduction

The recent advent of multicore technology in processor designs [1] has introduced parallel computing power to the desktop. Unlike the increase in clock frequency characteristic for previous generations of processors, application programs do not automatically benefit from multiple cores, but require explicit parallelisation. This need brings parallel and distributed programming techniques from the niche of traditional supercomputing application areas into the main stream of software engineering. However, the shift in application characteristics also demands new programming concepts, tools and infrastructure.

We present the design of the declarative coordination language S-NET [2]. S-NET³ describes the coordination behaviour of networks of asynchronous components and their orderly interconnection via typed streams. We deliberately restrict S-NET to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes* in S-NET terminology, to conventional languages. For the time being, we focus on the data-parallel functional array programming language SAC [3] as our primary box language. However, coordination and computation layers are sufficiently orthogonalised in our approach to support a range of box implementation languages. In fact, the same S-NET may well contain boxes implemented in different box languages. This strict separation between computing and coordination layers facilitates the reuse of existing software and opens an avenue towards mixed language programming.

An S-NET box is connected to the outside world by two typed streams, a single input stream and a single output stream. Data on these streams is organised as non-recursive records, i.e. collections of label-value pairs. The operational behaviour of a box is characterised by a stream transformer function that maps a single record from the input stream to a (possibly empty) stream of records on the output stream. In order to facilitate dynamic reconfiguration of networks, a box has no internal state and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network.

³ <http://snet.feis.herts.ac.uk/>

Boxes execute fully asynchronously: as soon as a record is available on the input stream, a box may start computing and producing records on the output stream. The restriction to a single input stream avoids any confusion as whether to implicitly synchronise or how to respond to partially available data.

S-NET features two built-in components: Filter boxes take care of various kinds of house-keeping tasks that do not require the full power of a box language, e.g. deletion, duplication or renaming of record fields. Synchronisation boxes recombine multiple records on the input stream into a single record on the output stream based on structural pattern matching.

The construction of streaming networks based on instances of user-defined and built-in asynchronous components is a distinctive feature of S-NET: Thanks to the restriction to a single-input/single-output stream component interface we can describe entire networks through algebraic formulae. Network combinators either take one or two operand components and construct a network that again has a single input stream and a single output stream. As such a network again is a component, construction of streaming networks becomes an inductive process. We have identified a total of four network combinators that prove sufficient to construct a large number of network prototypes: static serial and parallel composition of heterogeneous components as well as dynamic serial and parallel replication of homogeneous components.

Structural subtyping on records greatly facilitates adaptation of individual components to varying contexts. More precisely, components only need to be specific about record fields that are actually needed for the associated computation or that are (at least potentially) created by that computation. In excess to these required fields, however, an input record to some component may have an arbitrary number of further fields. These additional fields bypass the component and are added to any outgoing record through an automatic coercion mechanism, named *flow inheritance*.

The remainder of the paper is organised as follows. In Section 2 we sketch out the S-NET type system. Sections 3 and 4 introduce boxes and networks, respectively. In Section 5 we discuss some related work, and we conclude in Section 6.

2 The type system of S-Net

2.1 Record types

The type system of S-NET is based on non-recursive variant records with *record subtyping*. Informally, a *type* in S-NET is a non-empty set of anonymous *record variants* separated by vertical bars. Each record variant is a possibly empty set of named *record entries*, enclosed in curly brackets. We distinguish two different kinds of record entries: *fields* and *tags*. A field is characterised by its *field name* (label); it is associated with an opaque value at runtime. Hence, fields can only be generated, inspected or manipulated by using an appropriate box language. A tag is represented by a name enclosed in angular brackets. At runtime tags are associated with integer values, which are visible to both box language code and S-NET. The rationale of tags lies in controlling the flow of records through a network. They should not be misused to hold box language data that can be represented as integer values.

We illustrate S-NET types by a simple example from 2-dimensional geometry: For instance, we may represent a rectangle by the S-NET type

$$\{x, y, dx, dy\}$$

providing fields for the coordinates of a reference point (x and y) and edge lengths in both dimensions (dx and dy). Likewise, we may represent a circle by the center point coordinates and its radius:

$$\{x, y, radius\}$$

Using the S-NET support for variant record types we may easily define a type for geometric bodies in general, encompassing both rectangles and circles:

```
{x, y, dx, dy} | {x, y, radius}
```

Often it is convenient to name variants. In S-NET this can be done using tags:

```
{<rectangle>, x, y, dx, dy} | {<circle>, x, y, radius}
```

S-NET also supports abstractions on types; we refer the interested reader to [2] for details.

2.2 Record subtyping

S-NET supports structural subtyping on record types. Subtyping essentially is based on the subset relationship between sets of record entries. Informally, a type is a subtype of another type if it has additional record entries in the variants or additional variants. For example, the type

```
{<circle>, x, y, radius, colour}
```

representing coloured circles is a subtype of the previously defined type

```
{<circle>, x, y, radius} .
```

Likewise, we may add another type to represent triangles:

```
{<rectangle>, x, y, dx, dy}
| {<circle>, x, y, radius}
| {<triangle>, x, y, dx1, dy1, dx2, dy2};
```

which again is a supertype of

```
{<rectangle>, x, y, dx, dy}
| {<circle>, x, y, radius}
```

as well as a supertype of

```
{<circle>, x, y, radius, colour} .
```

Our definition of record subtyping coincides with the intuitive understanding that a subtype is more specific than its supertype(s) while a supertype is more general than its subtype(s). In the first example, the subtype contains additional information concerning the geometric body (i.e. its colour) that allows us to distinguish, for instance, green circles from blue circles, whereas the more general supertype identifies all circles regardless of their colour. In our second example, the supertype is again more general than its subtype as it encompasses all three different geometric bodies. Subtype `{<circle>,x,y,radius,colour}` is more specific than its supertypes because it rules out triangles and rectangles from the set of geometric bodies covered. Unlike subtyping in object-oriented languages like C++ or Java our definition of record subtyping is purely structural; `{}` (i.e. the empty record) denotes the most common supertype.

2.3 Type signatures

Type signatures describe the stream-to-stream transformation performed by a box or a network. Syntactically, a type signature is a non-empty set of type mappings each relating an *input type* to an *output type*. The input type specifies the records a box or network accepts for processing; the output type characterises the records that the box or network may produce in response. For example, the type signature

```
{a,b} | {c,d} -> {<x>} | {<y>} , {e} -> {z}
```

describes a network that accepts records that either contain fields `a` and `b` or fields `c` and `d` or field `e`. In response to a record of the latter type the network produces records containing the field `z`. In all other cases, it produces records that either contain tag `x` or tag `y`.

2.4 Flow inheritance

Up-coercion of records upon entry to a certain box or network creates a subtle problem in the stream-processing context of S-NET. In an object-oriented setting the control flow eventually returns from a method invocation that causes an up-coercion. While during the execution of the specific method the object is treated as being one of the respective superclass, it always retains its former state in the calling context. In a stream-processing network, however, records enter a box or network through its input stream and leave it through its output stream, which are both connected to different parts of the whole network. If an up-coercion results in a loss of record entries, this loss is not temporary but permanent.

The permanent loss of record entries is neither useful nor desirable. For example, we may have a box that manipulates the position of a geometric body regardless of whether it is a rectangle, a circle or a triangle. The associated type signature of such a box could be as simple as $\{x,y\} \rightarrow \{x,y\}$. This box would accept circles, rectangles and triangles focusing on their common data (i.e. the position) and ignoring their individual specific fields and tags. Obviously, we must not lose this data as a consequence of the automatic up-coercion of complete geometric bodies to type $\{x,y\}$. Hence, we complement this up-coercion with an automatic down-coercion. More precisely, any field or tag of an incoming record that is not explicitly named in the input type of a box or network bypasses the box or network and is added to any outgoing record created in response, unless that record already contains a field or tag with the same label. We call this coercion mechanism *flow inheritance*.

As an example, let us assume a record $\{\langle \text{circle} \rangle, x, y, \text{radius}\}$ hits a box $\{x,y\} \rightarrow \{x,y\}$. While fields x and y are processed by the box code, tag `circle` and field `radius` bypass the box without inspection. As they are not mentioned in the output type of the box, they are both added to any outgoing record, which consequently forms a complete specification of a circle again.

3 Box abstractions

3.1 User-defined boxes

From the perspective of S-NET boxes are the atomic building blocks of streaming networks. Boxes are declared in S-NET code using the key word `box` followed by a box name as unique identifier and a box signature enclosed in round brackets. The box signature very much resembles a type signature with two exceptions: we use round brackets instead of curly brackets, and we have exactly one type mapping that has a single-variant input type. For example,

```
box foo ((a,b,<t>) -> (a,b) | (<t>));
```

declares a box named `foo`, which accepts records containing (at least) fields `a` and `b` plus a tag `t` and in response produces records that either contain fields `a` and `b` or tag `t`. Boxes are implemented using a box language rather than S-NET. It is entirely up to the box implementation to decide how many output records a box actually emits and of which of the output variants they are. This may well depend on the values of the input record entries and, hence, can only be determined at runtime.

Box signatures use round brackets rather than curly brackets to express the fact that in box signatures sequence does matter. (Remember that type signatures are true sets of mappings between true sets of record entries.) Sequence is essential to support a mapping to function parameters of some box language implementation rather than using inefficient means such as string matching of field and tag names. For example, we may want to associate the above box declaration `foo` with a C language implementation in the form of the C function `foo` shown in Fig. 1.

```

snet_handle_t *foo( snet_handle_t *handle, int *a, mytype_t *b, int t)
{
    /* some computation on a, b and t */
    snetout( handle, 1, a, b);
    /* some computation */
    snetout( handle, 2, t);
    return( handle);
}

```

Fig. 1. Example box function implementation in C

The entries of the input record type are effectively mapped to the function parameters in their order of appearance in the box signature. We implement record fields as opaque pointers to some data structure and tags as integer values. In addition to the box-specific parameters the box function implementation always receives an opaque S-NET handle, which provides access to S-NET internal data.

Since boxes in S-NET generally produce a variable number of output records in response to a single input record, we cannot exploit the function's return value to determine the output record. Instead, we provide a special function `snetout` that allows us to produce output records during the execution of the box function. The first argument to `snetout` is the internal handle that establishes the necessary link to the execution environment. The second argument to `snetout` is a number that determines the output type variant used. So, the first call to `snetout` in the above example refers to the first output type variant. Consequently, the following arguments are two pointers. The second call to `snetout` refers to the second output type variant and, hence, a single integer value follows. Eventually, the box function returns the handle to signal completion to the S-NET context.

This is just a raw sketch of the box language interfacing. Concrete interface implementations may look differently to accommodate characteristics of certain box languages, and even the same box language may actually feature several interface implementations with varying properties.

3.2 Filter boxes

The filter box in S-NET is devoted to housekeeping operations. Effectively, any operation that does not require knowledge of field values can be expressed by this versatile built-in box in a simpler way than using an atomic box and a fully-fledged box language implementation. Among these operations are

- elimination of fields and tags from records,
- copying fields and tags,
- adding tags,
- splitting records,
- simple computations on tag values.

Syntactically, a filter box is enclosed in square brackets and consists of a type (pattern) to the left of an arrow symbol and a semicolon-separated sequence of filter actions to the right of the arrow symbol, for example:

```
[{a,b,<t>} -> {a} ; {c=b,<u=42>} ; {b,<t=t+1>}]
```

This filter box accepts records that contain fields `a` and `b` as well as tag `t`. In general, the type-like notation to the left of the arrow symbol acts as a pattern on records; any incoming record's type must be a subtype of the pattern type.

As a response to each incoming record, the filter box produces three records on its output stream. The specifications of these three records are separated by semicolons to the right

of the arrow symbol. Outgoing records are defined in terms of the identifiers used in the pattern. In the example, the first output record only contains the field `a` adopted from the incoming record (plus all flow-inherited record entries). The second output record contains field `b` from the input record, which is renamed to `c`. In addition there is a tag `u` set to the integer value 42. The last of the three records produced contains the field `b` and the tag `t` from the input record, where the value associated with tag `t` is incremented by one. S-NET supports a simple expression language on tag values that essentially consists of arithmetic, relational and logical operators as well as a conditional expression.

3.3 Synchrocells

The synchrocell is the only “stateful” box in S-NET. It also provides the only means in S-NET to combine two existing records into a single one, whereas the opposite direction, the splitting of a single record, can easily be achieved by both user-defined boxes and built-in filter boxes. Syntactically, a synchrocell consists of an at least two-element comma-separated list of type patterns enclosed in `[|` and `|]` brackets, for example

```
[| {a,b,<t>}, {c,d,<u>} |]
```

The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single one that is released to the output stream. Matching here means that the type of the record is a subtype of the type pattern. The pattern also acts as an input type specification: a synchrocell only accepts records that match at least one of the patterns.

A synchrocell has storage for exactly one record of each pattern. When a record arrives at a fresh synchrocell, it is kept in this storage and is associated with each pattern that it matches. Any record arriving thereafter is only kept in the synchrocell if it matches a previously unmatched pattern. Otherwise, it is immediately sent to the output stream. As soon as a record arrives that matches the last remaining previously unmatched variant, all stored records are released. The output record is created by merging the fields of all stored records into the last matching record. If an incoming record matches all patterns of a fresh synchrocell right away, it is immediately passed to the output stream.

Although we called synchrocells “stateful” above, this is only true as far as individual records are concerned. Synchrocells nevertheless realise a functional mapping from input stream to output stream as a whole.

4 Streaming networks

4.1 Network definitions

User-defined and built-in boxes form the atomic building blocks for stream processing networks; their hierarchical definition is at the core of S-NET. As a simple example of a network definition take:

```
net X {
  box foo ((a,b)->(c,d));
  box bar ((c)->(e));
}
connect foo..bar;
```

Following the key word `net` we have the network name, in this case `X` and an optional block of local definitions enclosed in curly brackets. This block may contain nested network definitions and box declarations. Hierarchical network definitions incur nested scopes, but in the absence of relatively free variables the scoping rules are straightforward.

A distinctive feature of S-NET is the fact that complex network topologies are not defined by some form of wire list, but by an expression language. Each network definition contains such a topology expression following the key word `connect`. Atomic expressions are made up of box and network names defined in the current scope as well as of built-in filter boxes and synchronocells. Complex expressions are inductively defined using a set of network combinators that represent the four essential construction principles in S-NET: serial and parallel composition of two (different) networks as well as serial and parallel replication of one network, as sketched out in Fig. 2. Note that any network composition again yields a network with exactly one input and one output stream.

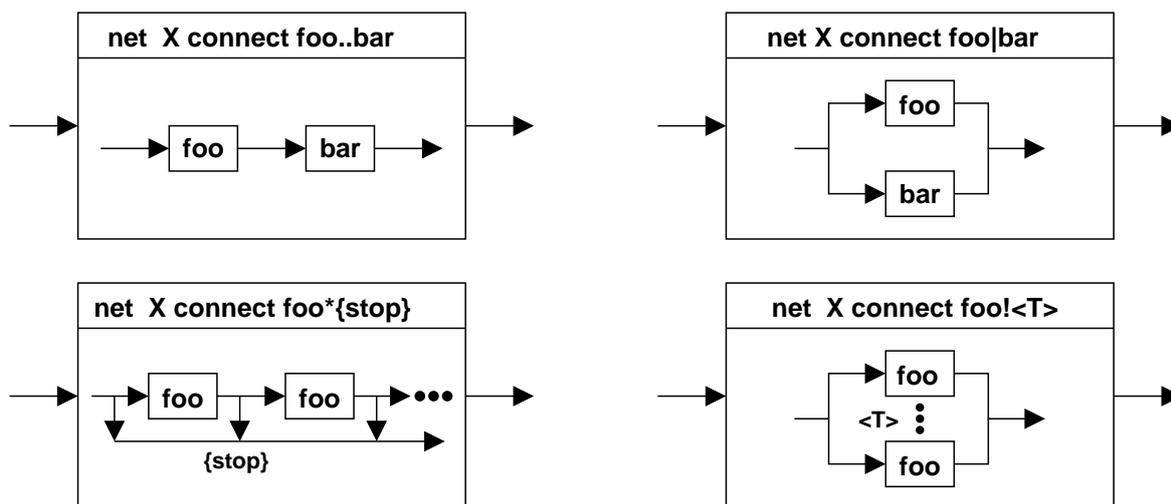


Fig. 2. Illustration of network combinators and their operational behaviour: serial composition (top-left), parallel composition (top-right), serial replication (bottom-left) and indexed parallel replication (bottom-right)

4.2 Serial composition

The binary serial combinator “`..`” connects the output stream of the left operand to the input stream of the right operand. The input stream of the left operand and the output stream of the right operand become those of the combined network. The serial combinator establishes computational pipelines, where records are processed through a sequence of computational steps.

In the example of Fig. 2, the two boxes `foo` and `bar` are combined into such a pipeline: all output from `foo` goes to `bar`. This example nicely demonstrates the power of flow inheritance: In fact the output type of box `foo` is not identical to the input type of box `bar`. By means of flow inheritance, any field `d` originating from box `foo` is stripped off the record before it goes into box `bar`, and any record emitted by box `bar` will have this field be added to field `e`. In contrast to box declarations, type signatures of networks are inferred by the compiler. For example the inferred type signature of the network `X` in the above example is `{a,b}->{d,e}`.

4.3 Parallel composition

The binary parallel combinator “`|`” combines its operands in parallel. Any incoming record is sent to exactly one operand depending on its own type and the operand type signatures. The output streams of the operand networks (or boxes) are merged into a single stream,

which becomes the output stream of the combined network. Fig. 2 illustrates the parallel composition of two networks `foo` and `bar` (i.e. `foo|bar`).

To be precise, any incoming record is sent to that operand network whose type signature's input type is matched best by the record's type. Let us assume the type signature of `foo` is `{a}->{b}` and that of `bar` is `{a,c}->{b,d}`. An incoming record `{a,<t>}` would go to box `foo` because it does not match the input type of box `bar`, but thanks to record subtyping does match the input type of box `foo`. In contrast, an incoming record `{a,b,c}` would go to box `bar`. Although it actually matches both input types, the input type of box `bar` scores higher (2 matches) than the input type of box `foo` (1 match). If a record's type matches both operand type signatures equally well, the record is non-deterministically sent to one of the operand networks.

4.4 Serial replication

The serial replication combinator “`*`” replicates the operand network (the left operand) infinitely many times and connects the replicas by serial composition. The right operand of the combinator is a type (pattern) that specifies a termination condition. Any record whose type is a subtype of the termination type pattern (i.e. matches the pattern) is released to the combined network's output stream.

In fact, an incoming record that matches the termination pattern right away is immediately passed to the output stream without being processed by the operand network at all. This coincidence with the meaning of star in regular expressions particularly motivates our choice of the star symbol. Fig. 2 illustrates the operational behaviour of the star combinator for a network `foo*{<stop>}`: Records travel through serially combined replicas of `foo` until they contain a tag `<stop>`. Actual replication of the operand network is demand-driven. Hence, networks in S-NET are not static, but generally evolve dynamically, though in a restricted way.

4.5 Indexed parallel replication

Last but not least, the parallel replication combinator “`!`” takes a network or box as its left operand and a tag as its right operand. Like the star combinator, it replicates the operand, but connects the replicas using parallel rather than serial composition. The number of replicas is conceptually infinite. Each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the given tag. Hence, all records that have the same tag value will be routed to the same replica of the operand network. Fig. 2 illustrates the operational behaviour of indexed serial replication for a network `foo!<T>`. In analogy to serial replication, instantiation of replicas is demand-driven.

4.6 Putting it all together

The restriction of every box and every network to a single input stream and a single output stream allows us to describe complex streaming networks in a very concise way using algebraic formulae rather than wire lists. Fig. 3 demonstrates the power of our approach by means of an example network

```
net XYZ connect ((A..B|C..D)!<i>)*{<stop>} .
```

The example uses 4 predefined boxes: `A`, `B`, `C` and `D`. Sequential compositions of `A/B` and `C/D`, respectively, are combined in parallel. The resulting subnetwork is replicated vertically through indexed parallel replication and, thereafter, horizontally through serial replication. Although Fig. 3 demonstrates the complexity of this network, its specification takes no more than half a line of code.

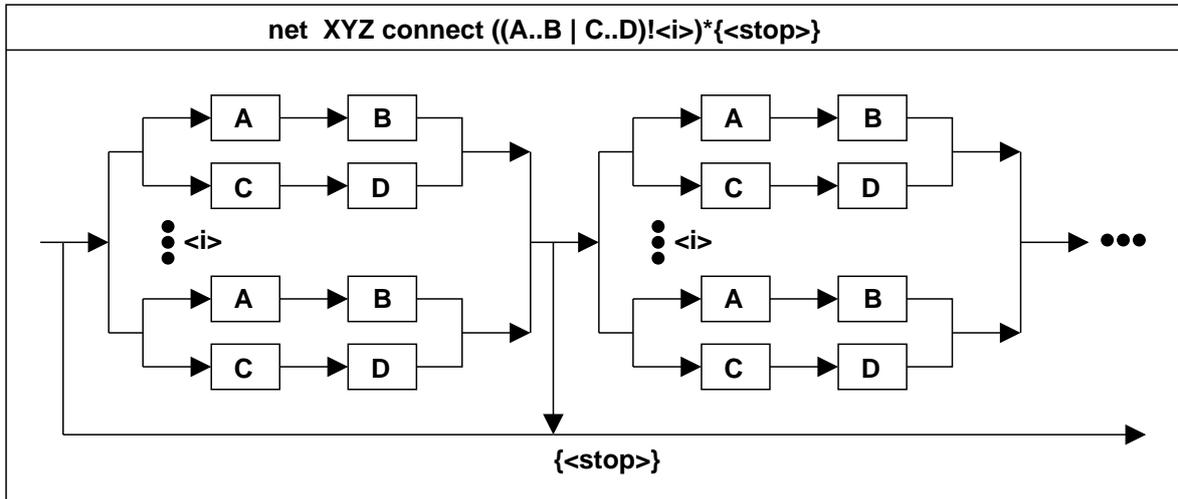


Fig. 3. Example of complex network construction with S-NET network combinators

5 Related work

The concept of stream processing has a long history (see [4] for a survey). The view of a program as a set of processing blocks connected by a static network of channels goes back at least as far as Kahn’s seminal work [5] and the language Lucid [6]. Kahn introduced the model of infinite-capacity, deterministic process networks and proved that it had properties useful for parallel processing. Lucid was apparently the first language to introduce the basic idea of a block that transforms input sequences into output sequences.

In the 1980s, a host of synchronous dataflow languages sprouted, notably Lustre [7] and Esterel [8]. They introduced explicit recurrence relations over streams and further developed the concept of synchronous networks. These languages are still being used for programming reactive systems and signal processing algorithms today [9]. The authors of Lustre broadened their work towards what they termed synchronous Kahn networks [10, 11], i.e functional programs where the connection between functions, although expressed as lists, is in fact ‘listless’: as soon as a list element is produced, the consumer of the list is ready to process it, so that there is no queue and no memory management required.

Likewise, Hume [12] combines the expressive power of Haskell-like box specifications with synchronous data flow processing. The emphasis of this project lies in the inference of exact bounds on resource consumption for applications in embedded systems.

A non-functional interpretation of Kahn networks is also receiving attention. The latest stream processing language of this category is, to the best of our knowledge, StreamIt [13].

The coordination aspect of S-NET is related to a large body of work in data-driven coordination, see [14] for a survey. Unlike many data-driven approaches, S-NET achieves *complete* separation of coordination and computation. This is achieved by using opaque SISO stream transformers implemented in a separate box language chosen by the programmer.

The earliest related proposal, to our knowledge, is the coordination language HOPLa from the Utrecht University’s Ariadne project [15]. It is a Linda-like coordination language, which uses record subtyping (which they call “flexible records”) in a manner similar to S-NET, but does not handle variants as we do, and has no concept of flow inheritance. Also, HOPLa has no static “wiring” and does not use the notion of type to establish concrete streaming network configurations.

Another early source to mention is the language SISAL [16], which pioneered high-performance functional array processing with stream communication. SISAL was not in-

tended as a coordination language, though, and no attempt at the separation of communication and computation was made.

Among more recent papers, we cite the work on the language Eden [17] as related to our effort, since it is based on the concept of stream communication. Here streams are lazy lists produced by processes defined in Haskell using explicit process abstraction and process instantiation. Unlike S-NET, Eden deploys processes fully dynamically and even allows completely dynamic channels. Eden has no provision for subtyping and does not integrate topology with types.

Another recent advancement in coordination technology is the language Reo [18]. Also based on streams, it concerns itself primarily with issues of channel and component mobility and does not exploit static connectivity or type-theoretical tools for network analysis.

We must also mention important theoretical advances in streaming networks. The key work in this area has been done by Stefanescu, who has developed several semantic models for streaming networks starting from flowcharts [19] and recently including models for non-deterministic stream processing developed collaboratively with Broy [20]. This work aims at an algebraic language for denotational semantics of stream processing. While as such it is not focused on pragmatic issues, it nevertheless offers important structuring primitives, which can be used as the basis for a network algebra.

6 Conclusions and future work

We have presented the design of S-NET, a declarative language for describing streaming networks of asynchronous components. Several features distinguish S-NET from existing stream processing approaches.

- S-NET boxes are fully asynchronous components communicating over buffered streams.
- S-NET thoroughly separates coordination aspects from computation aspects.
- The restriction to SISO components allows us to describe complex streaming networks by algebraic formulae rather than error-prone wiring lists.
- We utilise a type system to guarantee basic integrity of streaming networks.
- Data items are routed through networks in a type-directed way making the concrete network topology a type system issue.
- Record subtyping and flow inheritance make S-NET components adaptive to their environment.

The overall design of S-NET is geared towards facilitating the composition of components developed in isolation. The box language interface in particular allows existing code to be turned into an S-NET stream processing component with very little effort.

A prototype compiler for S-NET and a runtime system that maps S-NET boxes to Posix threads approach completion [21]. We are currently working on a non-trivial application from the plasma physics domain to demonstrate the suitability of S-NET to coordinate concurrent activities on a representative scale.

Acknowledgements

The development of S-NET is funded by the European Union through the FP-6 Integrated Project *ÆTHER* (Self-adaptive Embedded Technologies for Pervasive Computing Architectures), where S-NET has been adopted as the main high-level software approach.

References

1. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal* **30**(3) (2005)
2. Grelck, C., Shafarenko, A.: Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom (2006)
3. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming* **34**(4) (2006) 383–427
4. Stephens, R.: A survey of stream processing. *Acta Informatica* **34**(7) (1997) 491–541
5. Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, L., ed.: *Information Processing 74, Proc. IFIP Congress 74, August 5-10, Stockholm, Sweden, North-Holland* (1974) 471–475
6. Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. *Communications of the ACM* **20**(7) (1977) 519–526
7. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE* **79**(9) (1991) 1305–1320
8. Berry, G., Gonthier, G.: The estereel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* **19** (1992) 87–152
9. Binder, J.: Safety-critical software for aerospace systems. *Aerospace America* (2004) 26–27
10. Caspi, P., Pouzet, M.: Synchronous kahn networks. In Wexelblat, R.L., ed.: *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming.* (1996) 226–238
11. Caspi, P., Pouzet, M.: A co-iterative characterization of synchronous stream functions. In Bart Jacobs, Larry Moss, H.R., Rutten, J., eds.: *CMCS '98, First Workshop on Coalgebraic Methods in Computer Science Lisbon, Portugal, 28 - 29 March 1998.* (1998) 1–21
12. Hammond, K., Michaelson, G.: The design of hume: a high-level language for the real-time embedded systems domain. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers.* Volume 3016 of *Lecture Notes in Computer Science.*, Springer (2004) 127–147
13. Michael I. Gordon *et al*: A stream compiler for communication-exposed architectures. In: *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 2002.* (2002)
14. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. In: *Advances in Computers.* Volume 46. Academic Press (1998)
15. Florijn, G., Bessamusca, T., Greefhorst, D.: Ariadne and hopla: Flexible coordination of collaborative processes. In Ciancarini, P., Hankin, C., eds.: *First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, 15-17 April, 1996.* LNCS 1061. (1996) 197–214
16. Feo, J.T., Cann, D.C., Oldehoeft, R.R.: A report on the sisal language project. *J. Parallel Distrib. Comput.* **10**(4) (1990) 349–366
17. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. *Journal of Functional Programming* **15**(3) (2005) 431–475
18. Arbab, F.: Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.* **14**(3) (2004) 329–366
19. Stefanescu, G.: An algebraic theory of flowchart schemes. In Franchi-Zanettacci, P., ed.: *Proceedings 11th Colloquium on Trees in Algebra and Programming, Nice, France, 1986.* Volume LNCS 214., Springer-Verlag (1986) 60–73
20. Broy, M., Stefanescu, G.: The algebra of stream processing functions. *Theoretical Computer Science* (258) (2001) 99–129
21. Grelck, C., Penczek, F.: Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom (2006)