Concurrency Engineering with S-Net

Clemens Grelck[1,2], Sven-Bodo Scholz[2], and Alex Shafarenko[2]

[1] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl

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

Abstract. We present the design of S-Net, a coordination language and component technology based on stream processing. S-Net boxes integrate existing sequential code as stream-processing components into highly asynchronous concurrent streaming networks. Their construction is based on algebraic formulae built out of four network combinators. S-Net achieves a near-complete separation of concerns between application code, written in a conventional programming language, and coordination code, written in S-Net itself. 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 of 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. This shift demands new programming concepts, tools and infrastructure that are suitable for average application programmers not previously exposed to the pitfalls of concurrent program execution.

Parallel programming in the conventional style is considered notoriously difficult because it intertwines two different aspects od program execution: algorithmic behaviour, i.e. what is to be computed, and organisation of concurrent execution, i.e. how a computation is performed on multiple execution units including the necessary problem decomposition, communication and synchronisation. S-Net [2] is a novel declarative coordination language whose design thoroughly avoids the intertwining of computational and organisational aspects through active separation of concerns: S-Net completely separates the concern of writing sequential application building blocks (i.e. *application engineering)* from the the concern of composing these building blocks to form a parallel application (i.e.*concurrency engineering).*

More precisely, S-Net defines the coordination behaviour of networks of asynchronous, stateless 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. 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. 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 and a single output stream per box again again is motivated by separation of concurrency engineering from application engineering. If a box had multiple input streams, this would immediately raise the question as to what extent input data arriving on the various input streams is synchronised. Do we wait for exactly one data package on each input stream before we start computing like in Petri nets? Or do we alternatively start computing when the first data item arrives and see how far we get without the other data? Or could we even consume varying numbers of data packages from the various input streams? This immediately intertwines the question of synchronisation, which is a classical concurrency engineering concern, with the concept of the box, which in fact is and should only be an abstraction of a sequential compute component.

The same is true for the output stream of a box. Had a box multiple output streams, this would immediately raise the question of data routing, again a classical concurrency engineering concern, as the box code would need to decide to which stream data should be sent. Having a single output stream oinly, in contrast, clearly seperates the routing aspect from the computing aspect of the box and, thus, concurrency engineering from application engineering.

The construction of streaming networks based on instances of 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*.

To summarise, the motivation of S-Net is to completely separate algorithmic programming from concurrency engineering. Indeed any user-defined box represents an algorithm encapsulated in the form of a function that performs a computation on a data item (its argument list) and which computes and passes back to the environment one or more similar data items. Any communication or synchronisation actions, any division of work between workers and gathering of the data back in one place is happening in the coordination language. This makes boxes unit-testable in isolation, and also removes (due to the requirement of statelessness) any placement or mobility constraints from all user-defined boxes. This is in sharp contrast with SPMD programming styles inherent in MPI and similar parallel libraries, and to the best of our knowledge, any other coordination language.

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. We demonstrate their interaction by a small programming example in Section 5 and 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 supports type definitions; 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. In contrast, 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 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`}->{`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 {`<circle>,x,y,radius`} hits a box {`x,y`}->{`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.

```
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

Box signatures use round brackets instead of 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.

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, as demonstrated in Fig. 1. 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 synchrocells. 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.

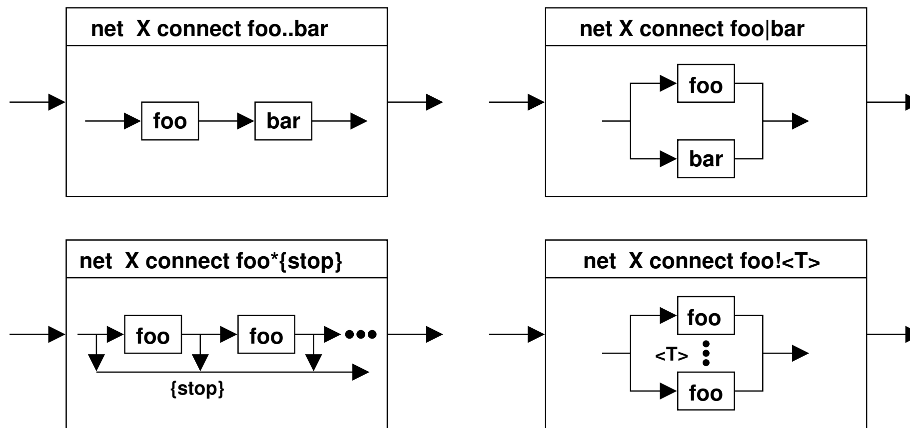

Fig. 2. Illustration of network combinators and their operational behaviour: serial composition (top-left), parallel composition (top-right), serial replication (bottom-left) and indexed parallel replication (bottom-right)

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 generally inferred by the compiler. For example the inferred type signature of the network X in the above example is {a,b}->{d,e}. Type inference is a particularly interesting aspect of S-NET. We refer the interested reader to [3] for a thorough treatment of the subject.

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 match a given type pattern, more precisely the type of the record is a record subtype of the specified type (pattern). Optionally, the exit pattern may be refined by a boolean expression on the values of the tags in the type pattern. 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.

Note that this construct in combination with serial replication allows dynamic, SPMD style connections: a network such as `(A!<P>)*<Y>` allows the box `A` to receive records with a certain value of `<P>` and create records with either the same or different value of `<P>` which will be fed to an appropriate replica of `A`. Any output from `A` that is meant to be released should be tagged with `<Y>`. It is quite obvious that dynamic communication could be made as complex as the programmer requires using more combinators, but crucially the only routing issue that is dealt with dynamically is *which* replica of a box a given record should be directed to, not which box, and since all replicas share the same type signature, S-Net remains type safe even under dynamic routing.

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.

To conclude this section, we wish to make a remark on the implementation. Space limitations do not permit us to touch on any details; however, the crucial point of S-Net implementation is the use of nondeterminism. Solutions for parallel computing tend to be deterministic since nondeterminism can affect the values being processed and can lead to incorrect results. However, in the context of stream processing nondeterminism manifests itself as the lack of order in a stream sequence. Obviously if the receiver box either does not need to receive the records in a certain order, or if the required order can be reconstructed at the output of the top-level network from the stream content, nondeterministic merges are safe. On the other hand, the use of nondeterministic merges dramatically reduces the latency of processing, since the implementation is free to

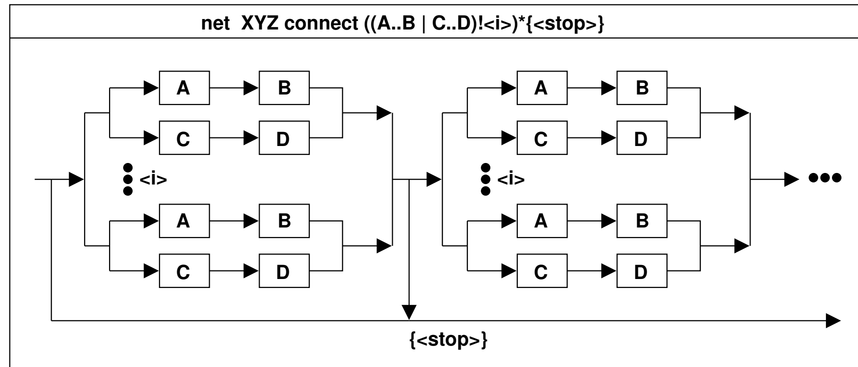

Fig. 3. Example of complex network construction with S-Net network combinators

merge streams in the order of arrival rather than queuing off records that have overtaken ones created earlier. Also in a multistage pipelined processing scheme, a record can represent work to be done by several algorithms in any order. For example, by a box A that determines the maximum row elements of a matrix and divides the rows by them, and by a box B that exchanges the left and right halves of all rows. A solution such as `X..((A..B)|(B..A))` where A and B present the same input type will be of this kind. The nondeterministic merger of the parallel combinator will be in a position to use its nondeterminism to choose the pathway depending on how busy boxes `A` and `B` are.

5 Example: solving Sudoku puzzles

We illustrate the potential of S-Net by a simple search problem: finding solutions to sudoku puzzles. While sudokus are simple enough to be explored in detail, they are computationally non-trivial as they require search over an imbalanced tree of theoretically up to 9^{81} possibilities. In this sense, sudokus act as an interesting, albeit simple, model of a real-world search problem.

Sudokus are played on a 9 by 9 board of numbers. Starting out from a board with several given numbers, the overall aim is to fill all empty positions with numbers so that the following conditions hold: (i) each row contains the numbers 1 to 9 exactly once, (ii) each column contains the numbers 1 to 9 exactly once, and (iii) each of the nine 3 by 3 sub-boards contains the numbers 1 to 9 exactly once. Although in general we may have an arbitrary number of solutions or no solution at all, well constructed sudokus have a unique solution.

Fig. 4 shows the S-Net implementation of a simple Sudoku solver; a textual specification of the same solver can be found in Fig. 5. The first box (`computeOpts`) expects records that only contain an abstract representation of a Sudoku board, i.e. the search problem to be solved. The box computes all potential settings for each open position in the Sudoku board (`opts`).

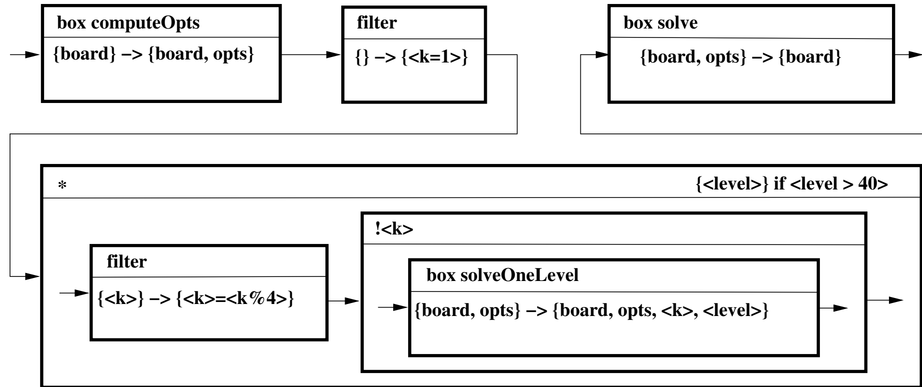

Fig. 4. Solving Sudoku puzzles with S-Net: graphical illustration

The solver itself is embedded within a serial replication: The box `solveOneLevel` tries to fix one further position on the board. For each possible number at that position it outputs a record containing the new board, the new (further) options and a tag `<level>`, whose value signals the number of fixed positions. In principle, the unfolding of the search tree could continue until all solutions are found, but on many target architectures it is more useful to control the amount of unfolding. In the example, we achieve this by refining the termination condition of the serial replication: as soon as 41 of the 81 positions of the board have been fixed, a record leaves the serial replication and is solved by the subsequent box `solve` without further unfolding of the S-Net network.

```
net sudoku {
  box computeOpts ((board)->(board,opts));
  box solveOneLevel ((board,opts)->(board,opts,<k>,<level>));
  box solve ((board,opts)->(board));
  net oneLevel connect [{<k>}->{<k=k%4>}] .. solveOneLevel!<k>;
}
connect computeOpts
        .. [{}->{<k=1>}]
        .. oneLevel * {<level>} if <level>40>
        .. solve;
```

Fig. 5. Solving Sudoku puzzles with S-Net: textual specification

The additional tag `<k>` in conjunction with indexed parallel replication creates another dimension of parallelism: Within each serial unfolding stage four concurrent instantiations of the `solveOneLevel` box process boards independently. A more thorough presentation of the Sudoku solver including box implementations in the functional array language SaC [4] can be found in [5].

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.

We have by now completed a prototype implementation of S-Net. This consists of a compiler for S-Net [6], including type inference [3], and box language interfaces for C and SaC [4] that allow us to write complete applications [5]. Furthermore, we have implemented a runtime system based on Posix threads for truly concurrent execution of S-Net programs on general-purpose shared memory multiprocessor and multicore architectures [7] as well as an MPI-based distributed memory runtime system for clusters of such machines [8].

Besides several smaller demonstrator applications we are currently working on a non-trivial plasma physics simulation as well as on a radar-based moving target identification (MTI) application that uses space-time adaptive processing (STAP) to demonstrate the suitability of S-Net to coordinate concurrent activities on a representative scale. In the future we aim at compiling S-Net to novel many-core processor designs like the MicroGrid architecture [9, 10] and to investigate into dynamic reconfiguration and self-adaptivity of S-Net networks [11].

References

1. Sutter, H.: The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobb's Journal **30** (2005)
2. Grelck, C., Shafarenko, A. (eds):, Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 1.0. Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2009)
3. Cai, H., Eisenbach, S., Grelck, C., Penczek, F., Scholz, S.B., Shafarenko, A.: S-Net Type System and Operational Semantics. In: Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland. (2008)

4. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
5. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)
6. 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 (2007)
7. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.
8. Grelck, C., Julku, J., Penczek, F.: Distributed S-Net. In Morazan, M., ed.: Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA, Seton Hall University (2009)
9. Bernard, T., Bousias, K., de Geus, B., Lankamp, M., Zhang, L., Pimentel, A., Knijnenburg, P., Jesshope, C.: A Microthreaded Architecture and its Compiler. In Arenez, M., Doallo, R., Fraguela, B., Tourino, J., eds.: Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS'06), Frankfurt/Main, Germany. (2006) 326–342
10. Bousias, K., Jesshope, C., Thiyagalingam, J., Scholz, S.B., Shafarenko, A.: Graph Walker: Implementing S-Net on the Self-adaptive Virtual Processor. In: Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland. (2008)
11. Penczek, F., Scholz, S.B., Grelck, C.: Towards Reconfiguration and Self-Adaptivity in S-Net. In Scholz, S.B., ed.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, Hertfordshire, UK. Technical Report 474, University of Hertfordshire, UK (2008) 330–339