The essence of synchronisation in asynchronous data flow programming

Grelck, C.

Publication date
2010
Document Version
Submitted manuscript
Published in
Technical Report - Department of Computer Science

The Essence of Synchronisation in Asynchronous Data Flow Programming *

Clemens Grelck

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

Abstract. We discuss the aspect of synchronisation in the language design of the asynchronous data flow language S-Net. Synchronisation is a crucial aspect of any coordination approach. S-Net provides a particularly simple construct, the synchrocell. The synchrocell is actually two simple to meet regular synchronisation demands itself. We show that in conjunction with other language feature, S-Net synchrocells can effectively do the job. Moreover, we argue that their simplistic design in fact is a necessary prerequisite to implement even more interesting scenarios, for which we outline ways of efficient implementation.

1 Introduction

The current hardware trend towards multi-core and soon many-core chip designs for increased performance creates a huge problem on the software side. Software needs to become parallel to benefit from future improvements in hardware, which will be rather in th number of cores available than in the individual computing power of a single core. So far, parallel programming has been confined to niches of software engineering, like for instance high performance computing. Now, parallel programming must become mainstream, and the real question is how existing software can be parallelised and how new software can be engineered without the cost currently attributed to, for instance, high performance computing.

S-Net [1–3] is a 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*, to conventional languages. An S-Net box is connected to the outside

* This work was supported by the European Union through the projects Æther and Advance.

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 only, in contrast, clearly separates 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.

Fig. 1 shows a simple example of an S-Net streaming network. The five boxes execute fully asynchronously and data items or tokens in the form of records flow through the stages of the streaming network. Neither the split point nor the merge point in the network constitute any form of synchronisation. Both

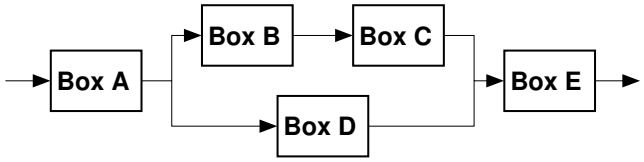

Fig. 1. Example of an S-Net streaming network of asynchronous components

are rather routing points where data can be either sent into multiple directions or received from multiple directions.

What does synchronisation mean in the streaming network context of S-Net? Due to the SISO restriction of network construction, no S-Net entity can have multiple input streams. Hence, synchronisation can only mean the combination of multiple data items or records appearing in some order on a single input stream. For this purpose S-Net provides a built-in component, the *synchrocell*. The design of the synchrocell is geared towards simplicity. In fact, a synchrocell waits for two differently typed records on its own input stream. Whichever record appears first is held within the synchrocell until the other appears as well. On this occasion the the synchrocell joins the two records and emits the resulting record on its output stream. Any synchrocell can only synchronise exactly once. Whenever a type pattern (more on this in the following chapters) has been matched any further records matching the same pattern will be forwarded to the output stream directly. Consequently, after successful synchronisation all patterns have been matched and, hence, the synchrocell becomes an identity box.

This operational behaviour may surprise. However, exactly this one-off behaviour proves essential to use synchrocells effectively in varying contexts. It is the contribution of this paper to show how and why.

The remainder of the paper is organised as follows: Section 2 provides more technical background information on S-Net while in Section 3 we focus entirely on the aspect of synchronisation and introduce S-Net synchrocells. Sections 4 and 5 describe two application scenarios for synchrocells that proves their simplistic design as an essential prerequisite. In Section 6 we sketch out directions of efficient implementation, and we conclude in Section 7.

2 S-Net in a Nutshell

S-Net is a high-level, declarative coordination language based on the concept of stream processing. As such S-Net promotes functions implemented in a standard programming language into asynchronously executed stream-processing components, termed *boxes*. Both imperative and declarative programming languages qualify as box implementation languages for S-Net, but we require any box implementation to be free of state on the coordination level. More precisely, a

box must not carry over any information between two consecutive activations on the streaming layer.

Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are organized as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. The fields are associated with values from the box language domain; they are entirely opaque to S-Net. Tags are associated with integer numbers, which are accessible both on the coordination and on the box level. Tag labels are distinguished from field labels by angular brackets.

Operationally, a box is triggered by receiving a record on its input stream. As soon as that happend, the box applies its box function to the record. In the course of function execution the box may communicate records on its output stream. Once the execution of the box function has terminated, the box is ready to receive and to process the next record on the input stream.

On the S-Net level 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,<e>));
```

declares a box that expects records with a field labeled `a` and a tag labeled `b`. The box responds with an unspecified number of records that either have just field `c` or fields `c` and `d` as well as tag `e`. The associated box function `foo` is supposed to be of arity two: the first argument is of type `void*` to qualify for any opaque data; the second argument is of type `int`.

The box signature naturally induces a *type signature*. Whereas a concrete sequence of fields and tags is essential for the proper specification of the box interface, we drop the ordering when reasoning about boxes in the S-Net domain. Consequently, this step turns tuples of labels into sets of labels. Hence, the type signature of box `foo` is `{a,<b>} -> {c} | {c,d,<e>}`. We call the left hand side of this type mapping the *input type* and the right hand side the *output type*, and we use curly brackets instead of round brackets to emphasise the set nature of types.

To be precise, this type signature makes `foo` accept *any* input record that has *at least* field `a` and tag `<b>`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$. Again, the variant v is a subtype of w if and only if every label $\lambda \in v$ also appears in w.

Subtyping on input types of boxes raises the question what happens to the excess fields and tags. As mentioned previously, S-Net supports the concept of *flow inheritance* whereby excess fields and tags 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 when it comes to getting boxes that were designed separately to work together in a streaming network.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property: any network, regardless of its complexity, is an SISO entity in its own right.

Let `A` and `B` denote two S-Net networks or boxes. Serial combination (`A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. As a consequence, `A` and `B` operate in pipeline mode.

Parallel combination (`A|B`) constructs a network where incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the combined network. The type system controls the flow of records. Each network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the subnetwork whose input type better matches the type of the record. If both branches match equally well, one is selected non-deterministically. Parallel composition can be used to route different kinds of records through different branches of the network (like branches in imperative languages) or, in the presence of subtyping, to create generic and specific alternatives triggered by the presence or the absence of certain fields or tags.

The parallel and serial combinators have their infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator `A*`*`type`* constructs an infinite chain of replicas of `A` connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. More precisely the type acts as a so-called *type pattern* and patter matching is defined via the same subtype relationship as defined above. Hence, a record leaves a serial replication context as soon as its type is a subtype of the type specified in the type pattern position.

The parallel replicator `A!<`*`tag`*`>` also replicates network `A` infinitely, but the replicas are connected in parallel. All incoming records must carry the tag; its value determines the replica to which a record is sent.

S-Net is an abstract notation for streaming networks of asynchronous components. It is a notation that allows programmers to express concurrency in an abstract and intuitive way without the need to reason about the typical annoyances of machine-level concurrent programming. Readers are referred to [3, 2, 4] for a more thorough presentation of S-Net and to [5, 6] for case studies on application programming with S-Net.

3 Synchronisation

What does synchronisation mean in the streaming network context of S-Net? Network combinators inspect records for routing purposes, but never manipulate individual records. This is the privilege of boxes and filters. They both may also split a single record into several records. However, no user-defined box or filter can ever join two records into a single one. The absence of state is an essential property of boxes. So, boxes cannot join records. Joining records is the essence of synchronisation in asynchronous data flow computing in the style of S-Net. Since synchronisation is an integral aspect of the coordination layer, we separate synchronisation as far as possible from any computing aspects and provide a special construct for this purpose: the *synchrocell*.

Syntactically, a synchrocell consists of two type patterns enclosed in `[|` and `|]` brackets, for example `[| {a,b,<t>}, {c,d,<u>} |]`. The synchrocell holds incoming records which match one of the patterns until both patterns have been matched. Only then are the records merged into a single one, which is released to the output stream. A match happens when the type of the record is a subtype of the type pattern. The pattern also acts as an input type specification of the synchrocell: it only accepts records that match at least one of the patterns. Any record arriving at a synchrocell whose type matches a pattern that has previously been matched by a preceding record, is directly forwarded to the output stream without any further processing or even analysis. Consequently, once both patterns of a synchrocell have been matched and a joined record has been emitted to the output stream, the synchrocell is bound to forward all further records regardless of their type directly to the output stream. Effectively, once a synchrocell has successfully synchronised and joined two records, it becomes the identity box. In a more operational sense, the synchrocell should be removed from the streaming network. In essence, synchronisation in S-Net is a one-shot operation.

This definition of synchronisation, in our view, is the bare minimum that is required: a one-shot synchronisation between two records on the same stream. This is truly the essence of synchronisation in the asynchronous data flow context of S-Net. This one-shot design of the synchrocell seems almost disturbing at first glance. However, in the following sections we will demonstrate how this simplistic design and only this design allows to implement essential higher level synchronisation features like continuous pairwise synchronisation or modelling of state.

4 Continuous Synchronisation

A very common form of synchronisation in a streaming network is *continuous synchronisation*, where the first record that matches pattern A is joined with the first record that matches pattern B, the second record that matches pattern A with the second record that matches pattern B and so on. With S-Net synchrocells this behaviour can easily be achieved when embedded into a serial

replication with an exit pattern that is the union of all patterns in the synchrocell. For example, the network

```
[| {A,B},{C,D} |] * {A,B,C,D}
```

achieves continuous synchronisation for records of type {A,B} with records of type {C,D}. Let us see how this works in detail. Fig. 2 illustrates the operational behaviour.

Initially the serial replication is uninstantiated. When the first record, say {A,B} arrives, it does not match the exit pattern of the star combinator and, hence, the serial replication unfolds and instantiates a fresh synchrocell, where the record is stored. Let us assume another record {A,B} comes next. Since it neither matches the exit pattern it is routed into the first instance of the serial replication and hits the synchrocell. As the corresponding pattern has already been matched, the synchrocell passes {A,B} on. It still does not match the exit pattern and, hence, a second instance of the serial replication unfolds, which again has a fresh synchrocell where the record is stored.

If the third record is of type {C,D}, it likewise is routed into the first instance of the serial replication where it hits the synchrocell that is already primed with the first record {A,B}. The synchrocell joins the two records forming a new record {A,B,C,D}. This record does match the exit pattern of the serial replication and leaves the network. Semantically the first synchrocell now becomes the identity, but, operationally, the whole first instance of the serial replication is removed and subsequent records immediately reach the second instance.

In fact, the combination of synchrocell and serial replication allows us to implement an unbounded matching store with the means of S-Net. Of course, we could also have just defined the semantics of the synchrocell as a matching store and provide continuous synchronisation built-in. We do not do so for two reasons. Firstly, it is good programming language design to keep the number and the complexity of primitive language constructs to the minimum and use constructive features whenever possible. Secondly, and actually more important, the simplistic design of synchrocells allows for another application, more precisely the modelling of state in an otherwise state free environment. We will learn more about this in the following section.

5 Modelling State

Functional programming and state typically do not go well together, and S-Net makes no exception here. In general, the absence of state must be seen as the great advantage of function approaches when it comes to parallel processing. In the context of S-Net, for example, the statelessness of boxes allows the S-Net runtime system to schedule box computations to computing resources at will, including the relocation and migration of computations between processing elements. A common example of modelling some form of state in main stream functional programming is tail-end recursion: a function with a number of parameters applies itself (recursively) to a new set of arguments computed from

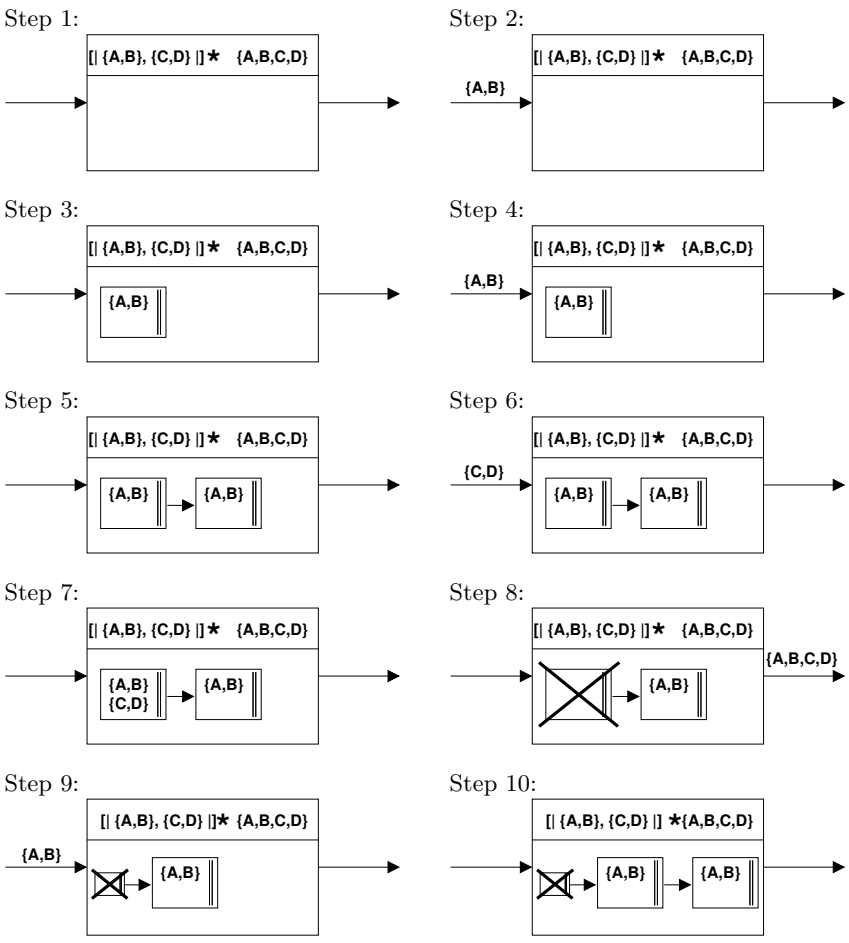

Fig. 2. Illustration of continuous synchronisation

the parameters in the last syntactic position, i.e. the value of recursive function application will become the value of the current function invocation.

Modelling state in S-Net follows the same basic idea, but, of course, has its intricacies due to the stream processing approach of S-Net. Fig. 3 shows an example implementation. The network `model_state` expects an initial state {`state`} followed by a sequence of values {`inval`} on its input stream; it will emit a sequence of values {`outval`} on its output stream, where each output value is a function of the corresponding input value and the accumulated state.

The local networks `join` and `id` wrap a synchrocell and a filter, respectively. Lifting these built-in constructs into separate networks is solely for illustration purposes and has otherwise no semantic implication. At the core of the network is the box `step` that expects a pair of state and value on the input stream. Based

```
net model_state {
  net join connect [|{state},{inval}|];
  net id connect [{inval} -> {inval}];
  box step ({state,inval} -> {state} | {outval});
}
connect (join..(id|step))*{outval};
```

Fig. 3. S-Net network that models a stateful computation.

on both the current state and the current value the box computes a new state and an output value that are individually emitted on the output stream. The network `model_state` is made up of a synchrocell that synchronises one state token with one input token. The subsequent parallel composition is crucial for the overall idea. Whenever the initial synchrocell combines state and value to one record, this record will be routed towards the `step` box because its input type is {`state,inval`}. Any subsequent value that just passes through the synchrocell will be routed towards the filter in the `id` network, which essentially is a typed identity.

The network consisting of `join`, `id` and `step` is embedded within a serial replication, i.e. the whole network is forward replicated on demand. The termination pattern {`outval`} ensures that any new value computed by the `step` box leaves the `model_state` network, whereas any new state computed by the `step` box triggers a re-instantiation of the network, including a fresh synchrocell where the new state is captured to wait for the corresponding value from the outermost input stream.

In Fig. 4 we illustrate the operational behaviour of the `model_state` network. Note that for space reasons we abbreviate {`state`}, {`inval`} and {`outval`} as {`s`}, {`iv`} and {`ov`}, respectively. Initially, the operand network of the serial replication remains uninstantiated, and only the appearance of the first state token (i.e. the initial state) triggers the instantiation of one instance. The {`s`} record is immediately captured in the synchrocell. Next, we expect the first value to appear on the input stream. It is likewise captured in the synchrocell leading to successful synchronisation and the construction of a joined {`s,iv`} record. The synchrocell becomes the identity thereafter.

The {`s,iv`} is routed towards the `step` box due to its type, which matches the input type of that box more than the input type of the `id` network, which is {`iv`}. The `step` box first emits an output value {`ov`}, which is sent to the global output stream, as it perfectly matches the termination pattern of the serial replication. Next, the `step` box emits a state token {`s`}. As this record does not match the termination pattern, it triggers a subsequent instantiation of the serial replication's operand network. The state token flows into this newly instantiated network, where is stuck in the fresh synchrocell.

In the meantime a second value appears on the input stream. It passes the first (disabled) synchrocell and bypasses the first instance of the `step` box. Note that we deliberately omit the identity filter in Fig. 4 as it can easily be identified

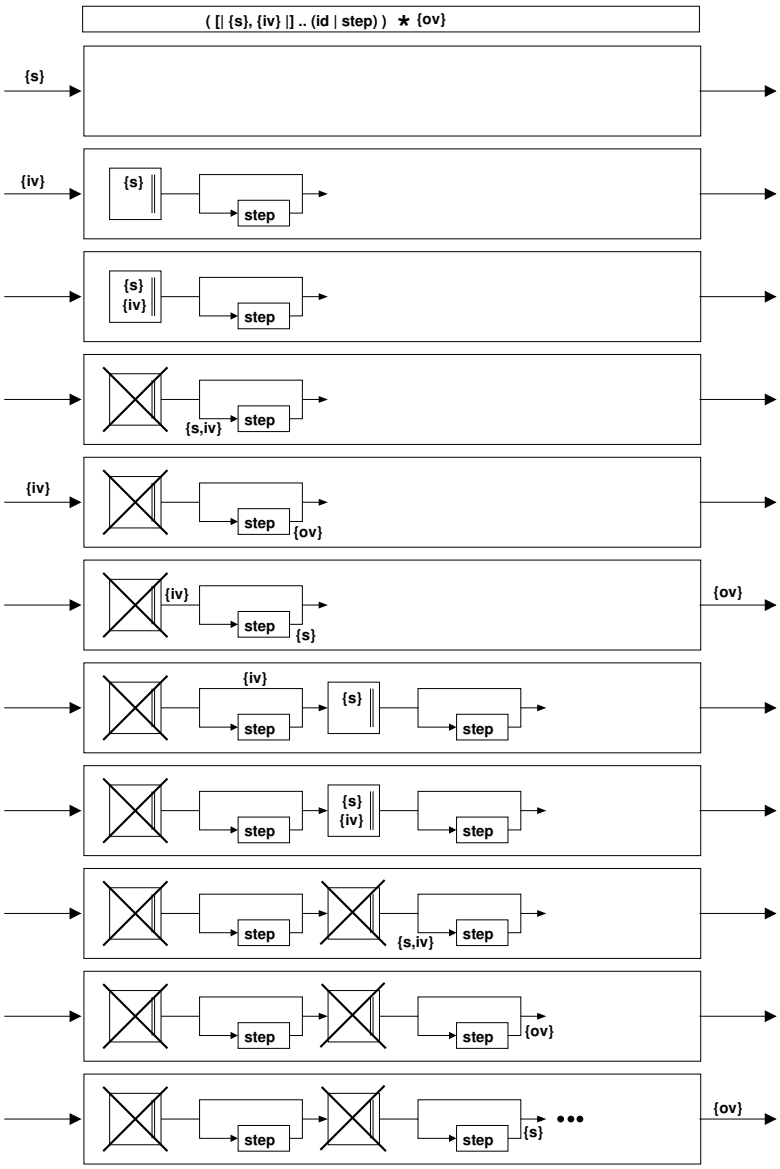

Fig. 4. Illustration of state-modelling network

by our implementation and simply leads to a bypass channel at runtime. Since {iv} does again not match the termination pattern of the serial replication, it is also routed into the second instance of the operand network. Here, it joins the state token {s} in the synchrocell. Now, history repeats itself leading to a

new output record sent to the global output stream and a new state record that triggers the next instantiation of the operand network.

The trick is that each instance of the subnetwork containing `join`, `id` and `step` processes exactly one input value and one instance of the state. After that each instance effectively becomes the identity for any subsequent input value. Conversely, each input record is routed such that it eventually reaches the front or active instance where it triggers the computation of a new output value and a new state.

6 Implementation Aspects

The current S-Net implementation for shared address space parallel architectures [7] is based on threads for boxes, including filters and synchrocells, and bounded FIFO buffers as streams from which threads read and to which threads write. In addition to the runtime components that explicitly appear in an S-Net specification, i.e. essentially boxes, filters and synchrocells, the S-Net runtime system makes use of a small further number of components that only implicitly appear on the S-Net level. These are routing components for parallel composition (routing based on best match of record type and the types of the outgoing streams), serial replication (routing into the replicated network potentially trigger a re-instantiation or routing outside) and parallel replication (routing based on the value of a named tag in the record) as well as merge components for all three underlying network combinators. In essence each network combinator except for serial composition inflicts the instantiation one routing and one merging component for network control.

The most naive implementation of synchrocells in this context is by an internal finite state machine that once it reaches its final state routes any further record on the input stream directly to the output stream. While obviously satisfying the semantics of S-Net [8], this solution is likewise obviously dissatisfying as the synchrocell component infinitely binds resources at runtime for no good and also the movement of records through the network is nothing but delayed by such useless components.

In fact, the current runtime system [7] takes a more reasonable approach. As soon as the finite state machine inherent to the synchrocell (runtime component) reaches its final state, it wraps the address of its input stream in a control message and sends this message to its own output stream. Thereafter, the component immediately terminates thus releasing all associated resources. When the subsequent component, i.e. the component that has the synchrocell's output stream as input stream, receives this control message, it releases its input stream, which in this situation is guaranteed to be empty, and uses the former synchrocell's input stream as its new input stream.

This implementation is adequate from the perspective of a single synchrocell, but is it also sufficient for the two application scenarios sketched out in the previous sections?

Let us first look at continuous synchronisation as outlined in Section 4. The runtime configuration of a synchrocell embedded within a serial replication is as sketched out in Fig. 5. As an example, we see a two-fold instantiation of the continuous synchronisation subnetwork. The serial repliation combinator leads to a sequence of routing components alternating with synchrocells as the only component embedded within the serial replication.

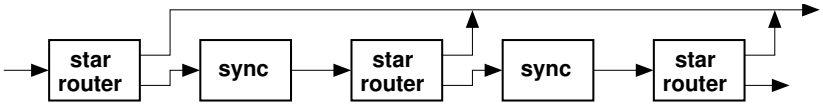

Fig. 5. Runtime configuration of continuous synchronisation

If a synchrocell terminates in this configuration, one routing component becomes directly connected to the next routing component. Since all routing component base their routing decision on the same type pattern, it is clear that serial replication with an empty operand network is semantically transparent, i.e. the instance of the serial replication may be eliminated and not just the synchrocell inside.

This situation can relatively easily be identified by the runtime system. Components register themselves with a stream when they first connect. More precisely, they store their individual identifiers withing the stream. At runtime, this allows components to identify their communication partners. Hence, a serial replication routing component, upon receiving its new input stream from the disabled synchrocell, can immediately identify whether or not the writing component to this stream belongs to the same serial replication. If so, the routing component, rather than reconnecting itself, forwards the stream to its own successor before terminates itself. As a consequence of this simple solution, any useless stage of the serial replication terminates and releases all resources along with the successful synchronisation.

The modelling of state, as proposed in the previous Section, is a much harder problem when it comes to efficient implementations. In this case, effective removal of the synchrocell or its bypassing does not suffice because for each stage of the serial replication the parallel composition within would simply remain as it is. This case is more complicated than that of continuous synchronisation, because the body of the serial replication contains a number of entities and not just a synchrocell. The trick in this case is that following successful synchronisation, the initial synchrocell can no longer produce records of the joint type, but only records of type. Consequently, all such records are guaranteed to use the bypass around the `step` box. Like in the continuous synchronisation case, the entire instance of the serial replication becomes the identity. However, this fact is much harder to identify here.

One potential approach goes as follows. When a synchrocell terminates, it not only wraps its input buffer into a control message but also a runtime representation of the synchrocell patterns. We assume that after termination of the synchrocell only records that match one of the synchronisation patterns may appear, but no records that contain the union of fields and tags. In the given example, this message arrives at the routing component derived from the parallel composition. If the routing component compares the type information contained in the message with the types that form the basis of its routing decision, it can quickly come to the conclusion that all further records must take the upper or bypass branch. Hence, the routing component can send a deactivation control message into the other branch that subsequently removes the whole (now useless) network until the corresponding join point.

Once this has happened we find the network in the same situation as in the previous scenario of continuous synchronisation. We have a serial replication with an empty body, i.e. two routing components of the star combinator have become immediate neighbours in the streaming network. Fortunately, we can use the same technique as above to finalise an entire stage in the dynamically replicated pipeline.

In either scenario the implementation tricks sketched out lead to a queue-like behaviour of serial replication. The pipeline grows at the front, and it shrinks at the end. Under normal circumstances, resource consumption should remain within reasonable bounds rather than grow unbounded as in the naive implementation.

7 Conclusion

In this paper we have explained and motivated the concept of synchronisation through synchrocells in the asynchronous data flow coordination language S-Net. Synchrocells, in our view, capture the bare essence of synchronisation in the context of S-Net. Through two examples, continuous synchronisation and the modelling of state, we demonstrate that it is this bare-bone design of synchrocells that acts as a prerequisite for the expressiveness of S-Net, when combined with other language features. In other words the deliberate restriction to a built-in synchronisation primitive that is unexpectedly simple proves to be essential for expressiveness, a seeming paradox.

We have shown how to use synchrocells to implement advanced synchronisation concepts. Their efficient implementation, however, is a slightly different matter. In fact, orthogonal language design that aims at using a minimum of built-in constructs of minimal complexity, puts a specific burden on language implementors. We have sketched out approaches to implement the scenarios used throughout the paper efficiently. Realisations of these implementation concepts are still outstanding leaving their practical evaluation as future work.

Acknowledgements

Design and implementation of S-Net is truly a team effort. The author would like to thank the other members of the S-Net team, namely Alex Shafarenko, Sven-Bodo Scholz and Frank Penczek, for endless fruitful discussions and an exciting research context.

References

1. Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18** (2008) 221–237
2. Grelck, C., Scholz, S.B., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
3. Grelck, C., Shafarenko, A. (eds):, Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2010)
4. Shafarenko, A.: Nondeterministic coordination using s-net. In Gentzsch, W., Grandinetti, L., Joubert, G., eds.: High Speed and Large Scale Scientific Computing. Volume 18 of Advances in Parallel Computing. IOS Press (2009) 74–96
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. Penczek, F., Herhut, S., Grelck, C., Scholz, S.B., Shafarenko, A., Barrere, R., Lenormand, E.: Parallel signal processing with S-Net. Procedia Computer Science **1** (2010) 2079 – 2088 ICCS 2010.
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, Revised Selected Papers. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.
8. Penczek, F., Grelck, C., Scholz, S.B.: An Operational Semantics for S-Net. In Chapman, B., Desprez, F., Joubert, G., Lichnewsky, A., Peters, F., Priol, T., eds.: Parallel Computing: From Multicores and GPU's to Petascale. Volume 19 of Advances in Parallel Computing. IOS Press (2010) 467–474