Implementing S-Net

A Typed Stream Processing Language

— Part I —

Compilation, Code Generation and Deployment

— DRAFT —

Version 0.3

December 8, 2006

Clemens Grelck and Frank Penczek

University of Hertfordshire

Department of Computer Science

Hatfield, Herts, AL10 9AB

United Kingdom

Abstract

We outline the architecture of an S-Net implementation consisting of compiler, code generator, deployer and runtime system. The S-Net compiler actually is a multi-stage compilation framework by itself. For this purpose we define a series of S-Net-like languages that gradually make the shift from full-fledged S-Net to a very simple distant variant of S-Net. The code generator takes this intermediate form to generate proper ISO C code with calls into a comprehensive runtime library. The deployer takes a fully compiled SNet and two interface descriptions for the global input and output stream and forms an executable program. The runtime system actually implements the various language features of S-Net and controls the setup of streaming networks and the orderly cooperative behaviour of asynchronous components. Different runtime system implementations address specific properties of destination architectures.

Contents

Chapter 1

System Architecture

1.1 Overview

S-Net [1, 2] is a declarative coordination language for describing streaming networks of asynchronous components on a high level of abstraction. Compiling S-Net specifications into efficiently executable code for various hardware architectures is a complex and demanding undertaking. Therefore, we systematically break down this task into smaller, as independent as possible units As illustrated in Fig. 1.1, we organise the system architecure of an S-Net implementation into four main functional units:

- compiler,
- code generator,
- deployer and
- runtime system.

The S-Net compiler takes an S-Net specification and, in a multi-stage process, transforms it into a substantially different textual representation. This representation is a radically simplified form of S-Net, extended by additional features supported in the runtime system. The S-Net code generator takes these internal textual representations and turns them into proper ISO C code with calls into the runtime system. Technically, the code generator may be considered the final stage of the compiler. However, for the sake of illustration we keep the two issues separate in this document.

The S-Net deployer combines a single (compiled) SNet[1] and two interface descriptions into an executable program. Furthermore, the deployer selects a concrete runtime system implementation.

[1] We use different fonts to distinguish between the language S-Net and the SNet networks it describes.

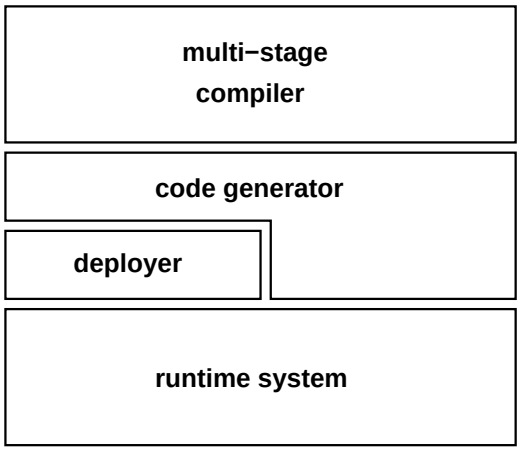

Figure 1.1: System architecture of S-Net implementation

The remainder of this document is organised as follows: The remaining sections of this chapter outline the principles of compiler, code generator, deployer and runtime system before we introduce a simple running example in Section 1.6. We will use this running example to illustrate the details of compiler, code generator, deployer and runtime system in Chapters 2, 3, 4 and 5, respectively.

1.2 Compiler architecture

In order to manage the complexity of compiling fully-fledged declarative S-Net code into a near machine-level representation we define several intermediate variants of S-Net. A multi-stage compilation framework gradually transforms S-Net specifications into less abstract and less declarative code.

Fig. 1.2 shows a sketch of the overall S-Net compiler architecture. We define five compilation stages: preprocessing, topology flattening, type inference, optimisation and postprocessing. In addition we have two auxiliary stages: parsing and printing. The five compilation stages share a common internal representation of SNets. The auxiliary stages transform textual representations of SNets into this internal representation (parser) and vice versa (printer).

The compilation process may start and stop in any of the compilation stages. The exact stage at which to start is determined by an identifier in the source code: the first line of text must contain a special comment of the form:

```
//!  snet code
```

If this identifier is not present, the compilation process starts at the very beginning. The final compilation stage is determined by a compiler flag. If that stage has been completed, the S-Net compiler prints the intermediate program representation to the standard output stream with the intermediate language identification properly set. The five intermediate languages, S-Net$_{\text{core}}$,

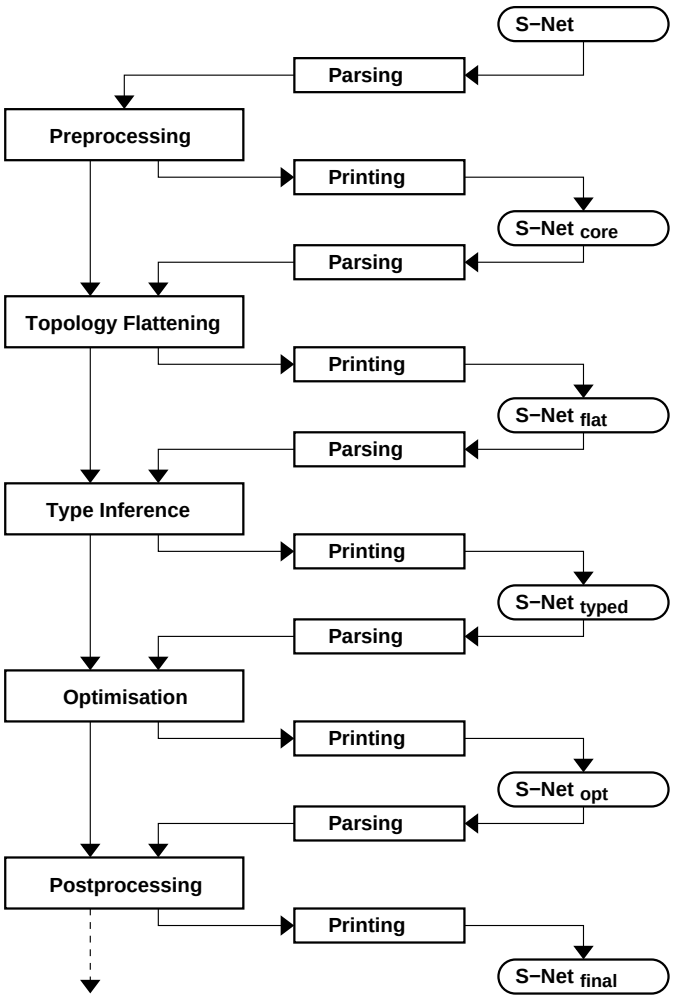

Figure 1.2: Architecture of S-Net compiler

S-Net$_{\text{flat}}$, S-Net$_{\text{typed}}$, S-Net$_{\text{opt}}$ and S-Net$_{\text{final}}$ are all variants of S-Net itself. Therefore, internal representation, parser and printer can be developed once and parameterised for the various intermediate languages.

The advantage of this multi-stage compiler architecture is that we may develop the individual parts mostly in isolation with well defined interfaces in between them. Ease of use is still achieved by the compiler driver that is responsible for user interaction and the orderly application of the individual compilation stages. Intermediate compiler phases may expect certain side conditions to hold in addition to the purely syntactical restrictions of the intermediate input language. In particular, conditions that have been checked, enforced or

created by preceding compiler phases are not to be checked again. If they are for some reason violated, a compiler phase may arbitrarily fail on the attempt to compile the erroneous code. The feature of stopping and resuming the compilation process is exclusively intended for the sake of compiler development and testing. In a product version it is to be deactivated or entirely removed.

1.3 Code generator architecture

The code generator essentially is the final stage of the S-Net compiler. However, as such it behaves differently from the other stages. It neither transforms the internal intermediate representation like the other stages nor does it merely print the internal representation into a textual format. To emphasise the special role of the code generator we describe it in a separate chapter.

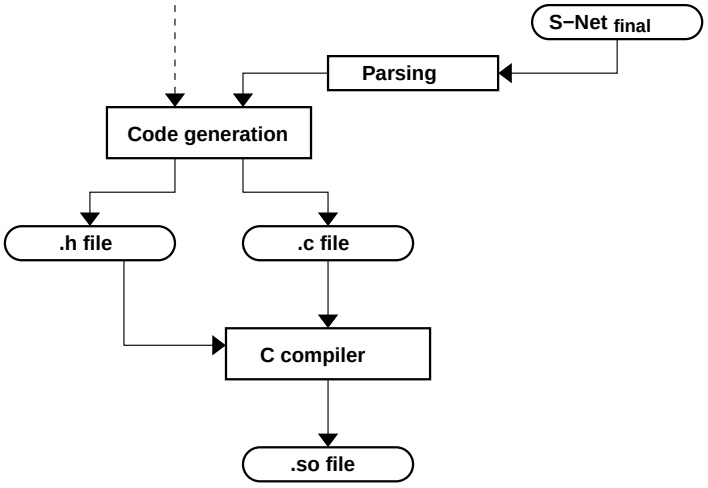

Figure 1.3: Architecture of S-Net code generator

Fig. 1.3 sketches out the architecture of the code generator. The code generator takes an SNet in the final intermediate representation and creates two files: a C source code file and a C header file. The header file contains numerical encodings for all field and tag names used throughout the SNet. Furthermore, it contains an external declaration of the generated function representing the exported network. They are needed for code generation whenever the given SNet is used in the context of another SNet. The C source code file, which includes the header file is then fed into an ISO C compiler that creates the final object code file, which will be taken by the S-Net deployer to form an executable SNet.

1.4 Deployer architecture

... yet to be developed in detail ...

1.5 Runtime system architecture

The runtime system is a rich library of system calls for runtime representations of types and patterns, for setting up SNets at runtime and for the dynamic control of asynchronous S-Net components and the communication channels between them.

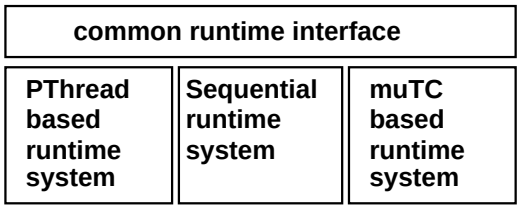

Figure 1.4: Architecture of S-Net runtime system

We show a sketch of the runtime system architecture in Fig. 1.4. The *common runtime interface* is an abstraction layer that allows us to support different target architectures without affecting the compilation and code generation process. For the time being, we envision three destination architectures:

- sequential execution,
- multithreading based on PThreads [3] and
- multithreading based on μTC [4].

The common runtime interface shields the specific properties of these and other target architectures from the S-Net compiler and code generator. Actually, changing the target architecure does not even require the recompilation of an SNet, but merely linking with a different runtime system implementation. Hence, the selection of a concrete target architecture is part of the deployment of SNets.

1.6 Running example

We illustrate both the compilation and the code generation process by means of the running example shown in Fig. 1.5. In order to incorporate as many as possible S-Net language features in a single example without making it overly complicated we use an abstract and artificial SNet rather than a concrete S-Net application.

```
type compAB_t = {A} -> {P}, {B} -> {Q};

net compABC ({A} | {C} -> {P}, {B} -> {Q}) {
  box compA ((A) -> (P));
  box compB ((B) -> (Q));
  box compC ((C) -> (P));
}
connect compA || compB || compC;

net example {
  net split
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine( (P,Q) -> (A,B) | (Y,Z));

  net compute {
    net compAB (compAB_t)
    connect compABC;

    net syncPQ
    connect [|{P},{Q}|] ** ({P,Q});
  }
  connect ( [{<T>} -> {}] .. compAB .. syncPQ ) !! (<T>);
}
connect (tag .. split .. compute .. examine) ** ({Y,Z});
```

Figure 1.5: Running example to illustrate compilation and code generation

The SNet `example` in Fig. 1.5 contains two top-level SNets: the auxiliary network `compABC` and the exported main network `example`. Whereas the former is a rather simple parallel composition of three boxes, the latter contains a serial composition of four subnetworks, `tag`, `split`, `compute` and `examine` embedded within a star combinator. Although this is not annotated in Fig. 1.5, we expect the network `example` to receive incoming records with fields `A` and `B`.

There is no definition of the network `tag`. So, we expect another file `tag.so` to contain the definition. Nevertheless, the idea of `tag` is to add a tag `T` to each record. The network `split`, which is made up of a single filter box, splits each record into two records, one containing field `A` and tag `T` and the other one containing field `B` and a copy of tag `T`.

The subnetwork `compute` does some computation on the data before the box `examine` checks incoming records for some termination condition. The latter either produces a new record {`A,B`} or a new record {`Y,Z`}. Given the termination condition of the star combinator in the connect expression of `example`, {`A,B`} records are directed into a new incarnation of the `tag..split..compute..examine` sequence while {`Y,Z`} records are directed to the global output stream.

The interior of the subnetwork `compute` is dynamically replicated using the parallel replication combinator (`!!`) based on the concrete values of tag `T`. Within, we first strip the tag `T` from each record as its sole purpose was to select the proper instance of this parallel replication. The actual computation is

performed by the subnetwork `compAB`. By providing a type signature for `compAB` we effectively specialise the top-level network `compABC` to only handle incoming records {`A`} and {`B`}, but not {`C`}. The resulting {`P`} and {`Q`} records are pairwise synchronised by a "starred" synchrocell in subnetwork `syncPQ`.

Chapter 2

Compilation

2.1 Parsing

The parser is effectively a 2-stage process, as sketched out in Fig. 2.1. For separation of concerns we distinguish between pure parsing and a separate dispatch phase. The pure parser reads in a textual S-Net specification and checks it for lexical and syntactical correctness. However, it does not do any further context checks. Pure parsing, hence, results in an internal representation that keeps all applied occurences of net or box names and, likewise, all occurences of record field names or tag names as plain character strings. In particular, the parser does not check context conditions such as the existence of a network definition for each instantiation. Such checks are deliberately organised into a separate dispatch phase to let the parser implementation focus on pure parsing issues.

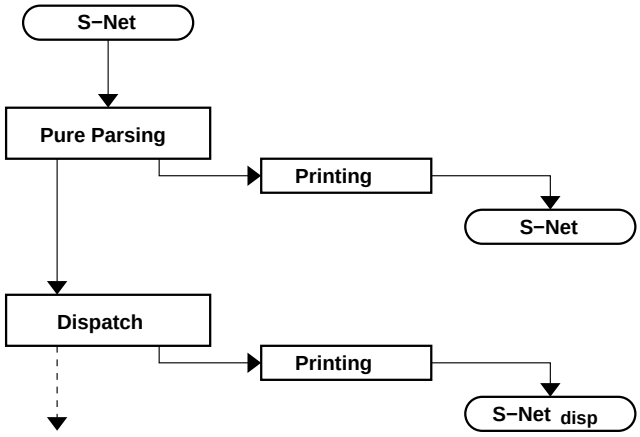

Figure 2.1: Architecture of parser

In analogy, to the overall compiler design, the compilation process may be terminated after pure parsing and the internal representation converted back into a textual specification. This feature allows us to verify the parsing step in isolation during the development process. Apart from source code comments, whitespace characters and text formatting issues this textual representation coincides with the original source S-Net specification.

$SNet_{disp}$	$\Rightarrow$	$\lceil Declaration \rceil^* \lceil Definition \rceil^*$
$Declaration$	$\Rightarrow$	**net** $NetName$ **;**
$Serial_{disp}$	$\Rightarrow$	**(** $SNetExpr\ SerialCombinator\ SNetExpr$ **)**
$Star_{disp}$	$\Rightarrow$	**(** $SNetExpr\ StarCombinator\ Terminator$ **)**
$Choice_{disp}$	$\Rightarrow$	**(** $SNetExpr\ ChoiceCombinator\ SNetExpr$ **)**
$Split_{disp}$	$\Rightarrow$	**(** $SNetExpr\ SplitCombinator\ Range$ **)**

Figure 2.2: Grammar of S-Net$_{\text{disp}}$

The dispatcher checks context conditions such as the existence of definitions for each instantiation of a locally defined net or box. At each such instantiation the dispatcher replaces the textual specification of the net or box by a reference to its definition. If the dispatcher does not find a matching local definition in the current scope, it considers the name to refer to an external SNet definition and creates a network declaration as stub code. Fig. 2.2 provides a formal definition of the intermediate language S-Net$_{\text{disp}}$. The sole differences with respect to S-Net proper are the network declaration part prior to the standard definitions and the resolution of combinator associativities and priorities in complex nested connect expressions by the parser: S-Net$_{\text{disp}}$ expects all topology definitions to be fully parenthesised.

Unlike all other compiler phases, which have a unique place in the overall compiler architecture, the parser may be used to process partially compiled intermediate code as well. In any case but parsing original S-Net source code, the dispatcher must expect external network declarations to be already present. If it still does not find a matching declaration for an otherwise unbound network identifier, the dispatcher issues an error message to properly report the detected inconsistency in the intermediate code.

In analogy to the storage of identifiers for boxes and networks, the dispatcher stores all record field names and tag names occuring in the SNet in a separate data base and replaces their names in all applied occurrences by a reference to the respective data base entry. Among others, these steps facilitate consistent renaming in the subsequent course of compilation.

References cannot be represented in textual representations properly. Therefore, we print plain names rather than references when returning to a textual representation of an SNet after dispatching. The need to restore references in the internal representation from names in the textual specification motivates us to integrate the dispatcher into the parser rather than into the preprocessor.

```
//! snet disp

net tag;

type compAB_t = {A} -> {P}, {B} -> {Q};

net compABC ({A} | {C} -> {P}, {B} -> {Q}) {
  box compA ((A) -> (P));
  box compB ((B) -> (Q));
  box compC ((C) -> (P));
}
connect (compA || (compB || compC));

net example {
  net split
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine( (P,Q) -> (A,B) | (Y,Z));

  net compute {
    net compAB (compAB_t)
    connect compABC;

    net syncPQ
    connect ([|{P},{Q}|] ** ({P,Q}));
  }
  connect (([{<T>} -> {}] .. (compAB .. syncPQ)) !! (<T>));
}
connect ((tag .. (split .. (compute .. examine)))**({Y,Z}));
```

Figure 2.3: Running example after parsing

The dispatch needs to be done whenever we convert a textual specification into the compiler-internal representation, regardless of where exactly we are in the overall compilation process otherwise.

Fig. 2.3 illustrates the effects of parsing (i.e. pure parsing and dispatch) on the running example, as introduced in Section 1.6. Those lines of code that show differences with respect to the original example are marked by a grey background. First, we observe the special comment in the first line of code that determines the state of the subsequent code with respect to the overall compilation process. It follows a network declaration for `tag` that is referred to in the connect expression of `example`, but nowhere defined in our example. The other difference we may observe are the additional parentheses in various connect expressions that make the implicitly defined associativities and priorities of combinators explicit.

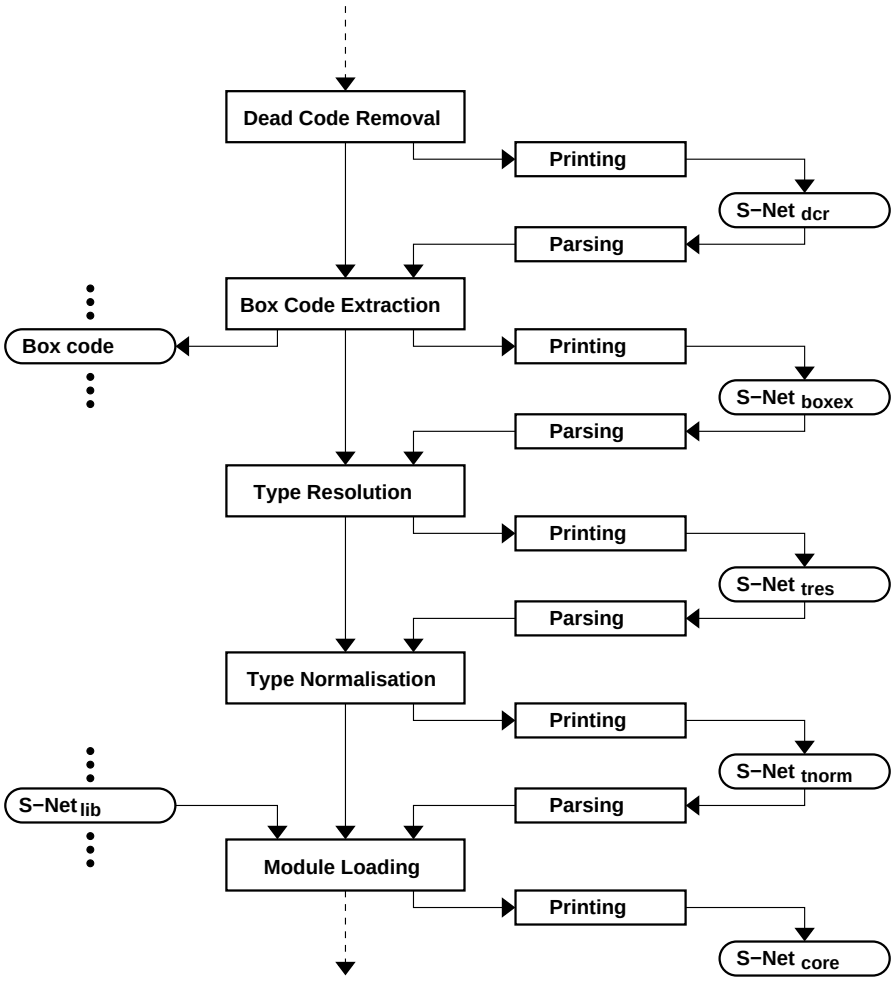

Figure 2.4: Architecture of preprocessor

2.2 Preprocessing

The preprocessor compiles S-Net$_{\text{disp}}$ code into the intermediate language S-Net$_{\text{core}}$. Effectively, it pursues a collection of tasks that are partly unrelated to each other, but share the common goal of making S-Net$_{\text{core}}$ a less complex language as compared with S-Net proper. Therefore, we adopt the overall compiler architecture for the preprocessor and organise the preprocessing stage again as a sequence of individual transformations with intermediate textual representations in between them. Fig. 2.4 gives an outline.

As a first step, we remove all those SNet definitions that are neither directly

nor indirectly referred to by the externally visible network definition that bears the same name as the file being compiled. Since this is a rather odd situation, we issue a warning to the programmer. The sole purpose of this transformation is to accelerate subsequent transformations by avoiding their application in useless situations. Dead Code Removal does (obviously) not change the intermediate representation. Nevertheless, we formally define the intermediate language S-Net$_{\text{dcr}}$ to distinguish intermediate code that has already undergone dead code removal from intermediate code that still needs to be cleaned up.

The second preprocessing step extracts inlined box language code from S-Net specifications. Such code is without inspection written to one external file per box language. Following this step the internal representation of SNets no longer allows for inlined box language code.

The next task of the preprocessor is the resolution of type definitions. Type definitions in S-Net (key word `type`) are nothing but placeholders for the respective definitions. Therefore, we replace each applied occurrence of a type name by the corresponding type definition.

Next, the normalisation of type representations means replacing type signatures with multi-variant input types by equivalent type signatures entirely made up of single-variant input types and additional type mappings.

$Declaration_{core}$	$\Rightarrow$	**net** *NetName* (*TypeSignature*) ;
$Definition_{core}$	$\Rightarrow$	*BoxDef* | *NetDef*
$BoxDef_{core}$	$\Rightarrow$	**box** *BoxName* (*BoxSignature*) ;
$TypeMapping_{core}$	$\Rightarrow$	*RecordType* `->` *Type*
$Type_{core}$	$\Rightarrow$	*RecordType* $\lceil$ `|` *RecordType* $\rfloor^*$

Figure 2.5: Grammar of S-Net$_{\text{core}}$

The module loader identifies external network declarations introduced by the dispatcher and identifies the corresponding compiled SNets searching in the current directory, in a directory path specified via a command line parameter on initiation of the compilation process and, eventually, in a similar path stored in an environment variable. If the module loader is unable to locate a compiled version of an external network, it issues an error message. Otherwise, it dynamically links with the with the corresponding S-Net library and calls a specific function from that library that recreates the internal representation of the network's type signature. This type signature is needed to continue with the compilation process.

Fig. 2.5 provides a formal definition of S-Net$_{\text{core}}$ as far as it differs from S-Net$_{\text{disp}}$. We see that due to the module loader network declarations now feature a type signature. Type definitions have vanished from the set of symbols that can be defined. Likewise, types may no longer use type names to refer to preceding type definitions. Last but not least, box definitions lack the potential

of inlined box language implementations

```
//! snet core

net tag ({A,B} -> {A,B,<T>});

net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {

  box compA( (A) -> (P));
  box compB( (B) -> (Q));
  box compC( (C) -> (P));
}
connect (compA || (compB || compC));

net example {
  net split
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine( (P,Q) -> (A,B) | (Y,Z));

  net compute {
    net compAB ({A} -> {P}, {B} -> {Q})

    connect compABC;

    net syncPQ
    connect ([|{P},{Q}|] ** ({P,Q}));
  }
  connect (([{<T>} -> {}] .. (compAB .. syncPQ)) !! (<T>));
}
connect ((tag .. (split .. (compute .. examine))) ** ({Y,Z}));
```

Figure 2.6: Running example after complete preprocessing

The aggregate effect of preprocessing on our running example can be seen in Fig. 2.6 We easily observe the new type signature in the declaration of the external network `tag` and the disappearance of the type definition of `compAB_t`.

2.3 Topology flattening

The topology flatter simplifies complex network topology specifications by systematically abstracting S-Net expressions into additional networks. Formally, the topology flatter turns S-Net$_{\text{core}}$ specifications into the S-Net$_{\text{flat}}$ intermediate representation format, defined in Fig. 2.7.

The significant difference between S-Net/S-Net$_{\text{core}}$ and S-Net$_{\text{flat}}$ is the restriction of operand networks of the four network combinators, serial, star, choice and split, to named networks. As a consequence, we may no longer represent nested topology expressions. The connect expression of any network may only contain a single instance of a box, a filter or a synchrocell. Alternatively, it may contain a single application of a network combinator to networks referred to by their names. In particular, each box, each filter and each synchrocell is

$SNetExpr_{flat}$	$\Rightarrow$	*BoxName*
	|	*NetName*
	|	*Sync*
	|	*Filter*
	|	*Combination*
$Serial_{flat}$	$\Rightarrow$	*NetName SerialCombinator NetName*
$Star_{flat}$	$\Rightarrow$	*NetName StarCombinator Terminator*
$Choice_{flat}$	$\Rightarrow$	*NetName ChoiceCombinator NetName*
$Split_{flat}$	$\Rightarrow$	*NetName SplitCombinator Range*

Figure 2.7: Grammar of S-Net$_{\text{flat}}$

embedded within its own network. This step prepares the internal representation of SNets for the subsequent type inference phase as we may now associate each level of a previously nested topology expression with its type signature.

Fig. 2.8 shows the impact of topology flattening on the running example. We observe the systematic recursive extraction of subexpressions from the network topology specifications of `compABC`, `example` and `compute` into the preceding contexts.

Topology flattening requires the introduction of new networks and, hence,

```
//! snet flat

net tag ({A,B} -> {A,B,<T>});

net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {
  box compA ((A) -> (P));
  box compB ((B) -> (Q));
  box compC ((C) -> (P));

  net _SL
  connect compA;

  net _SR {
    net _SL
    connect compB;

    net _SR
    connect compC;
  }
  connect _SL || _SR;
}
connect _SL || _SR;
```

Figure 2.8: Running example after topology flattening
(continued on next page)

```
net example {
  net split
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine( (P,Q) -> (A,B) | (Y,Z));

  net compute {
    net compAB ({A} -> {P}, {B} -> {Q})
    connect compABC;

    net syncPQ {
      net _ST
      connect [|{P},{Q}|];
    }
    connect _ST ** ({P,Q});

    net _IS {
      net _SL
      connect [{<T>} -> {}];

      net _SR
      connect compAB .. syncPQ;
    }
    connect _SL .. _SR;
  }
  connect _IS !! (<T>);

  net _ST {
    net _SR {
      net _SR {
        net _SR
        connect examine;
      }
      connect compute .. _SR;
    }
    connect split .. _SR;
  }
  connect tag .. _SR;
}
connect _ST ** ({Y,Z});
```

Figure 2.8: Running example after topology flattening
(continued from previous page)

new network names. The scheme for generation of network names must meet two requirements. Firstly, new network names must not collide with existing network names chosen by the programmer. In Fig. 2.8 all new network names start with an underscore letter; leading underscores are not permitted in language level S-Net identifiers. Secondly, the choice of name should facilitate the interpretation of flattened SNets with respect to the original specifications for a human reader. In Fig. 2.8 we use an algorithm that systematically creates names from the original location of a flattened network in the original topology

expression. The acronyms translate as described in Fig. 2.9.

SL	serial left
SR	serial right
PL	parallel left
PR	parallel right
ST	star
IS	index split

Figure 2.9: Creating network names from location in topology expression

2.4 Type inference

The type inference system, as the name suggests, infers a type signature for each S-NET$_{\text{flat}}$ network following the formal type rules presented in [2]. More formally, the type inference system turns S-NET$_{\text{flat}}$ code into S-NET$_{\text{typed}}$ code, as defined in Fig. 2.10. In fact, the syntactic differences between S-NET$_{\text{flat}}$ and S-NET$_{\text{typed}}$ are rather small: We only require that each network definition is associated with a type signature.

More precisely, the type inference system may infer a type in addition to a potentially programmer-supplied type signature or input type specification. In fact, existing type information requires the type inference system to do more than just inference. As explained in [2], the annotated type signature must be in box subtype relationship to the inferred type signature. Here, the type inference system effectively becomes a type checker and produces an error message if the condition is not met. If the type check succeeds, type inference continues with the annotated type rather than the inferred type.

$$
\begin{array}{lcl}
\mathit{NetDef}_{\mathit{typed}} & \Rightarrow & \textbf{net}\ \ \mathit{NetName}\ \mathit{NetTypes}\ \lceil\, \mathit{NetBody}\ \rfloor\ \mathit{Connect} \\
\mathit{NetTypes} & \Rightarrow & (\ \ \mathit{TypeSignature}\ \ )\ \ \lceil\ (\ \ \mathit{NetSignature}\ \ )\ \ \rfloor
\end{array}
$$

Figure 2.10: Grammar of S-NET$_{\text{typed}}$

Instead of a full type signature, S-NET allows the programmer to only annotate an input type to a network definition. If so, it must be in subtype relationship to the input type of the inferred type signature. Otherwise, we produce a type error message. Type inference continues with a type signature amalgamated from the inferred type signature and the annotated input type. If the annotated input type contains less variants than the input type of the inferred type signature, the additional type mappings are eliminated from the type signature. If a variant of the input type contains additional record entries compared with the corresponding variant of the input type of the inferred type signature, the additional record entries are added to the right hand side of the

corresponding type mapping. Last but not least, the output type of the annotated type signature may contain fewer record entries than the output type of the inferred type signature. In this case, the type inference system introduces appropriate filter boxes to adapt the internal (inferred) type signature to the annotated type signature.

If the inferred type signature is not identical to the one with which we continue type inference, the annotated type information is stored as an optional auxiliary type signature in the textual representation of S-Net$_{\text{typed}}$. Differences between annotated and inferred type information may be exploited for optimisation purposes at a subsequent compilation stage.

```
//! snet typed

net tag ({A,B} -> {A,B,<T>});

net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {
  box compA ((A) -> (P));
  box compB ((B) -> (Q));
  box compC ((C) -> (P));

  net _SL ({A} -> {P})
  connect compA;

  net _SR ({B} -> {Q}, {C} -> {P}) {
    net _SL ({B} -> {Q})
    connect compB;

    net _SR ({C} -> {P})
    connect compC;
  }
  connect _SL || _SR;
}
connect _SL || _SR;
```

Figure 2.11: Running example after type inference
(continued on next page)

Fig. 2.11 demonstrate the effect of type inference on the running example. Each and every network definition is now associated with a type signature. In the case of connect expressions that solely consist of the instance of a box, a filter or a sychrocell, the type signature may be derived rather straightforwardly from the box signature, the filter encoding or the synchronisation pattern, respectively. The other cases involving network combinator applications are handled as described in [2].

In the case of `compABC` the inferred type signature turned out to be identical to the annotated type signature. Hence, we store only one. The situation is different with `compAB`. Here, we infer a type signature that has one additional type mapping when compared to the annotated type signature. As a consequence, we keep both.

```
net example ({A,B} -> {Y,Z}) {
  net split ({A,B,<T>} -> {A,<T>} | {B,<T>})
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine ((P,Q) -> (A,B) | (Y,Z));

  net compute ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
    net compAB ({A} -> {P}, {B} -> {Q}, {C} -> {P})
               ({A} -> {P}, {B} -> {Q})
    connect compABC;

    net syncPQ ({P} -> {P,Q}, {Q} -> {P,Q}) {
      net _ST ({P} -> {P} | {P,Q}, {Q} -> {Q} | {P,Q})
      connect [| {P}, {Q}|];
    }
    connect _ST ** ({P,Q});

    net _IS ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
      net _SL ({<T>} -> {})
      connect [{<T>} -> {}];

      net _SR ({A} -> {P,Q}, {B} -> {P,Q})
      connect compAB .. syncPQ;
    }
    connect _SL .. _SR;
  }
  connect _IS !! (<T>);

  net _ST ({A,B} -> {A,B} | {Y,Z}) {
    net _SR ({A,B,<T>} -> {A,B} | {Y,Z}) {
      net _SR ({A,<T>} -> {A,B} | {Y,Z}, {B,<T>} -> {A,B} | {Y,Z}) {
        net _SR ({P,Q} -> {A,B} | {Y,Z})
        connect examine;
      }
      connect compute .. _SR;
    }
    connect split .. _SR;
  }
  connect tag .. _SR;
}
connect _ST ** ({Y,Z});
```

Figure 2.11: Running example after type inference
(continued from previous page)

2.5 Optimisation

The purpose of the optimiser is to reduce resource requirements and to improve the runtime performance of compiled SNets. For this purpose, the optimiser must make assumptions on both the deployer and on the runtime system. Optimisation naturally is a collection of independent and interdependent code transformations. Optimisation may always be considered optional and may (selectively) switched on and off for evaluation purposes.

There are two different categories of optimisations: Optimisations of the first category remain in the existing representational framework, i.e. transform S-Net$_{\text{typed}}$ code into semantically equivalent, but presumably more efficient, S-Net$_{\text{typed}}$ code. The second category of optimisations exploit additional features supported by the runtime system, but for resons of language design and software engineering principles not by the language S-Net itself. In order to support the second category of optimisations we define the additional intermediate language S-Net$_{\text{opt}}$ that, at the same time, forms the target language of the S-Net compiler and, likewise, the source language of the S-Net deployer. However, as optimisations are optional, S-Net$_{\text{opt}}$ is restricted to be a superset of S-Net$_{\text{typed}}$, i.e., any legal S-Net$_{\text{typed}}$ program automatically qualifies as an S-Net$_{\text{opt}}$ program.

$$
\begin{array}{lcl}
\mathit{Choice}_{opt} & \Rightarrow & \mathit{NetName}\ \mathit{ChoiceCombinator}\ \mathit{NetName} \\
 & | & \mathit{ChoiceCombinator}\ \mathit{NetName}\ \mathit{NetName}\ \lceil\, \mathit{NetName}\ \rfloor+
\end{array}
$$

Figure 2.12: Grammar of S-Net$_{\text{opt}}$

An example of the first category of optimisation is the systematic exploitation of differences between inferred and annotated type signatures for networks. Annotation of type signatures may be used to reduce the number of input variants, i.e., a network definition may cater for additional alternatives that may not be useful or needed in a certain instance of the network. By annotation of a type signature the programmer can make this information explicit, which in turn enables the optimiser to specialise such networks and to eliminate useless branches in the network definition. Another example of this category of optimisation is the coalascing of sequences of filters into a single, semantically equivalent filter.

An example of the second category of optimisation is the support for multiary parallel choice combinators in the S-Net runtime system. It is more efficient to analyse a record once and send it to one of multiple output channels than processing it by a cascade of binary choices. However, from a language design perspective the binary choice combinator is preferable because it is simpler and nesting easily provides the required expressiveness. Therefore S-Net$_{\text{opt}}$ and the intermediate languages thereafter feature n-ary versions of the two parallel choice combinators, as defined in Fig. 2.12. Essentially, the optimiser bridges the gap between language design and runtime performance issues.

Fig. 2.13 demonstrates the effect of some optimisations on the running example. Firstly, we note the 3-ary application of the choice combinator in the connect expression of `compABC`. Secondly, we observe the specialised variant of `compABC` as local redefinition within the body of `compAB`. Here, our optimisation exploits the fact that we effectively do not use the full flexibility of `compABC`. Furthermore, we could apply our dead code removal transformation once again and eliminate the original definition of `compABC` as it is not referred to anywhere

```
//! snet opt

net tag ({A,B} -> {A,B,<T>});

net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {
  box compA ((A) -> (P));

  box compB ((B) -> (Q));

  box compC ((C) -> (P));

  net _SL ({A} -> {P})
  connect compA;

  net _SR__SL ({B} -> {Q})
  connect compB;

  net _SR__SR ({C} -> {P})
  connect compC;
}
connect || _SL _SR__SL _SR__SR;

net example ({A,B} -> {Y,Z}) {
  net split ({A,B,<T>} -> {A,<T>} | {B,<T>})
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine ((P,Q) -> (A,B) | (Y,Z));
```

Figure 2.13: Running example after optimisation
(continued on next page)

else. However, we do keep it to illustrate the further processing of multi-ary choice combinators.

2.6 Postprocessing

In analogy to the preprocessor, the postprocessor again is a collection of several independent measures that prepare the internal representation of intermediate S-Net code for the final code generation step. Fig. 2.14 provides an outline of the individual phases.

The first postprocessing phase is a package translator. Whenever a record crosses the boundary between two S-Net packages (i.e. files), we need to translate the record field and tag names from the domain of one file (or unit of compilation) to the domain of the other file. We explicitly represent this translation in the intermediate code by introducing *translators* and package qualifiers for external symbols.

Fig. 2.15 illustrates the introduction of translators by an excerpt from our running example focussing on the instantiation of the external net `tag`. All symbols in the external network declaration of `tag` are now qualified by the

```
    net compute ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
      net compAB ({A} -> {P}, {B} -> {Q}) {
        net compABC ({A} -> {P}, {B} -> {Q}) {
          box compA ((A) -> (P));

          box compB ((B) -> (Q));

          net _SL ({A} -> {P})
          connect compA;

          net _SR__SL ({B} -> {Q})
          connect compB;
        }
        connect _SL || _SR__SL;
      }
      connect compABC;

      net syncPQ ({P} -> {P,Q}, {Q} -> {P,Q}) {
        net _ST ({P} -> {P} | {P,Q}, {Q} -> {Q} | {P,Q})
        connect [| {P}, {Q}|];
      }
      connect _ST ** ({P,Q});

      net _IS ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
        net _SL ({<T>} -> {})
        connect [{<T>} -> {}];

        net _SR ({A} -> {P,Q}, {B} -> {P,Q})
        connect compAB .. syncPQ;
      }
      connect _SL .. _SR;
    }
    connect _IS !! (<T>);

    net _ST ({A,B} -> {A,B} | {Y,Z}) {
      net _SR ({A,B,<T>} -> {A,B} | {Y,Z}) {
        net _SR ({A,<T>} -> {A,B} | {Y,Z},  {B,<T>} -> {A,B} | {Y,Z}) {
          net _SR ({P,Q} -> {A,B} | {Y,Z})
          connect examine;
        }
        connect compute .. _SR;
      }
      connect split .. _SR;
    }
    connect tag .. _SR;
  }
  connect _ST ** ({Y,Z});
```

Figure 2.13: Running example after optimisation
(continued from previous page)

package name `tag`. The translators themselves are embedded within a new wrapper network named `tag` (without package qualifier). All references to the external network `tag` in the file now point to this wrapper rather than to the

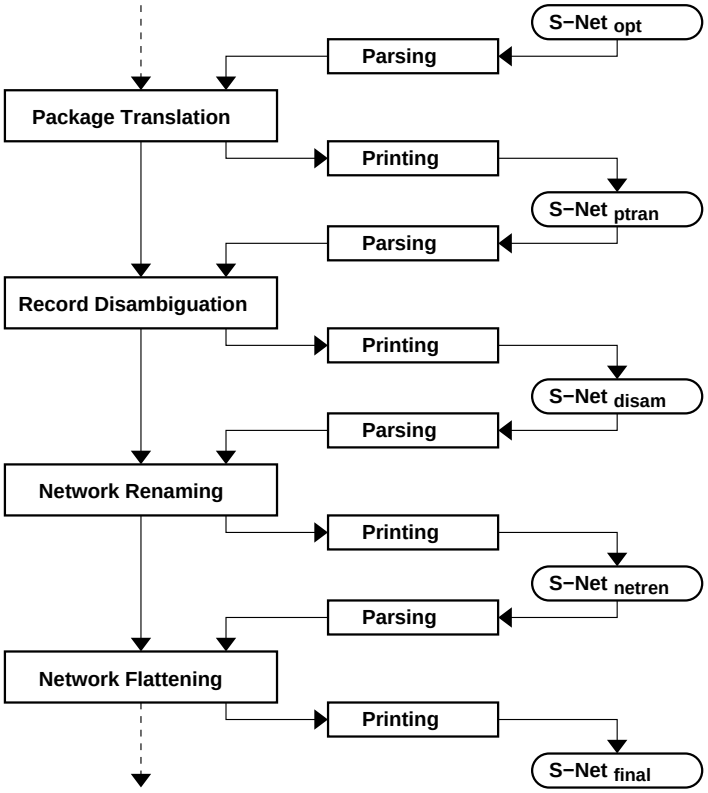

Figure 2.14: Architecture of postprocessor

external declaration. The wrapper network is the flattened and fully typed representation of the S-Net expression

```
[(A,B) -> (tag::A,tag::B)]
  .. tag::tag
     .. [(tag::A,tag::B,<tag::T>) -> (A,B,<T>)]
```

The syntax of the translators resembles that of filters. However, we use box types with round brackets rather than record types with curly brackets. This emphasises the significance of order as, for example in the first translator, we must map `A` to `tag::A` and `B` to `tag::B`. Fig. 2.16 provides a formal definition of the grammar of S-Net$_{\text{ptran}}$.

The second postprocessing step is a record entry disambiguation phase. The purpose of this phase is to separate unrelated occurrences of identical field and tag names by renaming. In particular, we must guarantee that the names of record entries that are flow-inherited at some level of our network tree are not used again at a deeper level. Whenever we detect such a case, we rename the

```
//! snet ptran

net tag::tag ({tag::A,tag::B} -> {tag::A,tag::B,<tag::T>});

net tag ({A,B} -> {A,B,<T>}) {
  net translate_in ({A,B} -> {tag::A,tag::B})
  connect [(A,B) -> (tag::A,tag::B)];

  net translate_out ({tag::A,tag::B,<tag::T>} -> {A,B,<T>})
  connect [(tag::A,tag::B,<tag::T>) -> (A,B,<T>)];

  net tag ({tag::A,tag::B} -> {tag::A,tag::B,<tag::T>})
  connect tag::tag;

  net in ({A,B} -> {tag::A,tag::B,<tag::T>})
  connect translate_in .. tag;
}
connect in .. translate_out;
```

Figure 2.15: Running example after package translation (excerpt)

nested occurrence of the name. This step is essential for the efficient implementation of flow inheritence. Since it requires a relatively complex setting of nested network specifications to create the need for record entry disambiguation, we refrain from an illustration in the course of our running example.

The third postprocessing phase consistently renames all networks to reflect the specific location of their definition within the overall tree structure of SNet definitions. Fig. 2.17 demonstrates the effect of network renaming on the running example. Essentially each network name is prefixed with the names of all networks in whose definitions it is embedded. The use of double underscores,

$Declaration_{ptran}$	$\Rightarrow$	**net** *Netname* `::` *NetName* `(` *TypeSignature* `)` `;`
$SNetExpr_{ptran}$	$\Rightarrow$	*BoxName*
	$\mid$	$[$ *Netname* `::` $]$ *NetName*
	$\mid$	*Sync*
	$\mid$	*Filter*
	$\mid$	*Translator*
	$\mid$	*Combination*
Translator	$\Rightarrow$	`[` *BoxType* `->` *BoxType* `]`
Field	$\Rightarrow$	$[$ *Netname* `::` $]$ *FieldName*
SimpleTag	$\Rightarrow$	`<` $[$ *Netname* `::` $]$ *TagName* `>`
BindingTag	$\Rightarrow$	`<` `#` $[$ *Netname* `::` $]$ *TagName* `>`

Figure 2.16: Grammar of S-Net$_{\text{ptran}}$

```
//! snet netren

net tag::tag ({tag::A,tag::B} -> {tag::A,tag::B,tag::<T>});

net tag ({A,B} -> {A,B,<T>}) {
  net tag__translate_in ({A,B} -> {tag::A,tag::B})
  connect [(A,B) -> (tag::A,tag::B)];

  net tag__translate_out ({tag::A,tag::B,tag::<T>} -> {A,B,<T>})
  connect [(tag::A,tag::B,tag::<T>) -> (A,B,<T>)];

  net tag__tag ({tag::A,tag::B} -> {tag::A,tag::B,<tag::T>})
  connect tag::tag;

  net tag__in ({A,B} -> {tag::A,tag::B,tag::<T>})
  connect tag__translate_in .. tag__tag;
}
connect tag__in .. tag__translate_out;

net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {
  box compA ((A) -> (P));
  box compB ((B) -> (Q));
  box compC ((C) -> (P));

  net compABC___SL ({A} -> {P})
  connect compA;

  net compABC___SR__SL ({B} -> {Q})
  connect compB;

  net compABC___SR__SR ({C} -> {P})
  connect compC;
}
connect || compABC___SL compABC___SR__SL compABC___SR__SR;

net example ({A,B} -> {Y,Z}) {

  net example__split ({A,B,<T>} -> {A,<T>} | {B,<T>})
  connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

  box examine ((P,Q) -> (A,B) | (Y,Z));
```

Figure 2.17: Running example after network renaming
(continued on next page)

which is ruled out in proper S-Net, to separate the various prefixes from each other and from the original name prevents name clashes. Not that we deliberately do not rename boxes because we consider box names to refer to some external implementation function whose name is fixed.

Network renaming is a preparation step for the final postprocessing phase: the transformation of the hierarchical network structure into a flat sequence of network and box definitions. Separation of renaming and restructuring fa-

```
  net example__compute ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
    net example__compute__compAB ({A} -> {P}, {B} -> {Q}) {
      net example__compute__compAB__compABC ({A} -> {P}, {B} -> {Q}) {
        box compA ((A) -> (P));
        box compB ((B) -> (Q));

        net example__compute__compAB__compABC___SL ({A} -> {P})
        connect compA;

        net example__compute__compAB__compABC___SR__SL ({B} -> {Q})
        connect compB;
      }
      connect example__compute__ compAB__compABC___SL
              || example__compute__compAB__compABC___SR__SL;
    }
    connect example__compute__compAB__compABC;

    net example__compute__syncPQ ({P} -> {P,Q}, {Q} -> {P,Q}) {
      net example__compute__syncPQ___ST ({P} -> {P} | {P,Q},
                                         {Q} -> {Q} | {P,Q})
      connect [| {P}, {Q}|];
    }
    connect example__compute__syncPQ___ST ** ({P,Q});

    net example__compute___IS ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
      net example__compute___IS___SL ({<T>} -> {})
      connect [{<T>} -> {}];

      net example__compute___IS___SR ({A} -> {P,Q}, {B} -> {P,Q})
      connect example__compute__compAB .. example__compute__syncPQ;
    }
    connect example__compute___IS___SL .. example__compute___IS___SR;
  }
  connect example__compute___IS !! (<T>);

  net example___ST ({A,B} -> {A,B} | {Y,Z}) {
    net example___ST___SR ({A,B,<T>} -> {A,B} | {Y,Z}) {
      net example___ST___SR___SR ({A,<T>} -> {A,B} | {Y,Z},
                                 {B,<T>} -> {A,B} | {Y,Z}) {
        net example___ST___SR___SR___SR ({P,Q} -> {A,B} | {Y,Z})
        connect examine;
      }
      connect example__compute .. example___ST___SR___SR___SR;
    }
    connect example__split .. example___ST___SR___SR;
  }
  connect tag .. example___ST___SR;
}
connect example___ST ** ({Y,Z});
```

Figure 2.17: Running example after network renaming
(continued from previous page)

```
//! snet final

net tag::tag ({tag::A,tag::B} -> {tag::A,tag::B,tag::<T>});

net tag__translate_in ({A,B} -> {tag::A,tag::B})
connect [(A,B) -> (tag::A,tag::B)];

net tag__translate_out ({tag::A,tag::B,tag::<T>} -> {A,B,<T>})
connect [(tag::A,tag::B,tag::<T>) -> (A,B,<T>)];

net tag__tag ({tag::A,tag::B} -> {tag::A,tag::B,<tag::T>})
connect tag::tag;

net tag__in ({A,B} -> {tag::A,tag::B,tag::<T>})
connect tag__translate_in .. tag__tag;

net tag ({A,B} -> {A,B,<T>})
connect tag__in .. tag__translate_out;

box compA ((A) -> (P));
box compB ((B) -> (Q));
box compC ((C) -> (P));

net compABC___SL ({A} -> {P})
connect compA;

net compABC___SR__SL ({B} -> {Q})
connect compB;

net compABC___SR__SR ({C} -> {P})
connect compC;

net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P})
connect || compABC___SL compABC___SR__SL compABC___SR__SR;

net example__split ({A,B,<T>} -> {A,<T>} | {B,<T>})
connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];

box examine ((P,Q) -> (A,B) | (Y,Z));

box compA ((A) -> (P));
box compB ((B) -> (Q));
```

Figure 2.18: Running example after complete postprocessing
(continued on next page)

cilitates the realisation and maintenance of both individual phases. Fig. 2.18 shows the final S-Net representation of our running example. This concludes the compilation process. From this representation we start the generation of ISO C code, as explained in Chapter 3. Fig. 2.19 summarises the complete grammar of S-Net$_{\text{final}}$.

```
net example__compute__compAB__compABC___SL ({A} -> {P})
connect compA;

net example__compute__compAB__compABC___SR__SL ({B} -> {Q})
connect compB;

net example__compute__compAB__compABC ({A} -> {P}, {B} -> {Q})
connect example__compute__ compAB__compABC___SL
        || example__compute__compAB__compABC___SR__SL;

net example__compute__compAB ({A} -> {P}, {B} -> {Q})
connect example__compute__compAB__compABC;

net example__compute__syncPQ___ST ({P} -> {P} | {P,Q},
                                   {Q} -> {Q} | {P,Q})
connect [| {P}, {Q}|];

net example__compute__syncPQ ({P} -> {P,Q}, {Q} -> {P,Q})
connect example__compute__syncPQ___ST ** ({P,Q});

net example__compute___IS___SL ({<T>} -> {})
connect [{<T>} -> {}];

net example__compute___IS___SR ({A} -> {P,Q}, {B} -> {P,Q})
connect example__compute__compAB .. example__compute__syncPQ;

net example__compute___IS ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q})
connect example__compute___IS___SL .. example__compute___IS___SR;

net example__compute ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q})
connect example__compute___IS !! (<T>);

net example___ST___SR___SR___SR ({P,Q} -> {A,B} | {Y,Z})
connect examine;

net example___ST___SR___SR ({A,<T>} -> {A,B} | {Y,Z},
                           {B,<T>} -> {A,B} | {Y,Z})
connect example__compute .. example___ST___SR___SR___SR;

net example___ST___SR ({A,B,<T>} -> {A,B} | {Y,Z})
connect example__split .. example___ST___SR___SR;

net example___ST ({A,B} -> {A,B} | {Y,Z})
connect tag .. example___ST___SR;

net example ({A,B} -> {Y,Z})
connect example___ST ** ({Y,Z});
```

Figure 2.18: Running example after complete postprocessing
(continued from previous page)

SNet	⇒	*⌈Declaration ⌉* ⌈Definition ⌉**
Declaration	⇒	**net** *Netname* `::` *NetName* `(` *TypeSignature* `)` `;`
Definition	⇒	*BoxDef* | *NetDef*
BoxDef	⇒	**box** *BoxName* `(` *BoxSignature* `)` `;`
BoxSignature	⇒	*BoxType* `->` *BoxType ⌈* `|` *BoxType ⌉**
BoxType	⇒	`(` *⌈RecordEntry ⌈* `,` *RecordEntry ⌉* ⌉* `)`
NetDef	⇒	**net** *NetName* `(` *TypeSignature* `)` *Connect*
TypeSignature	⇒	*TypeMapping ⌈* `,` *TypeMapping ⌉**
TypeMapping	⇒	*RecordType* `->` *RecordType ⌈* `|` *RecordType ⌉**
RecordType	⇒	`{` *⌈RecordEntry ⌈* `,` *RecordEntry ⌉* ⌉* `}`
RecordEntry	⇒	*Field* | *Tag*
Field	⇒	*⌈Netname* `::` *⌉ FieldName*
Tag	⇒	*SimpleTag* | *BindingTag*
SimpleTag	⇒	`<` *⌈Netname* `::` *⌉ TagName* `>`
BindingTag	⇒	`<` `#` *⌈Netname* `::` *⌉ TagName* `>`
Connect	⇒	**connect** *SNetExpr* `;`
SNetExpr	⇒	*BoxName*
	|	*⌈Netname* `::` *⌉ NetName*
	|	*Sync*
	|	*Filter*
	|	*Translator*
	|	*Combination*
Sync	⇒	`[|` *Pattern ⌈* `,` *Pattern ⌉+* `|]`
Pattern	⇒	*RecordType*
Filter	⇒	`[` *Pattern* `->` *⌈FilterOut ⌈* `;` *FilterOut ⌉* ⌉* `]`
	|	`[` `]`
FilterOut	⇒	`{` *⌈FilterOutField ⌈* `,` *FilterOutField ⌉* ⌉* `}`
FilterOutField	⇒	*Field ⌈* `<-` *FieldInit ⌉*
	|	*Tag ⌈* `<-` *TagInit ⌉*
FieldInit	⇒	*Field*
TagInit	⇒	*Tag*
	|	*IntegerConst*

Figure 2.19: Grammar of S-Net$_{\text{final}}$
(continued on next page)

Translator	⇒	`[` *BoxType* `->` *BoxType* `]`
Combination	⇒	*Serial* | *Star* | *Choice* | *Split*
Serial	⇒	*NetName SerialCombinator NetName*
Star	⇒	*NetName StarCombinator Terminator*
Terminator	⇒	`(` *Pattern* ⌈ `,` *Pattern* ⌉* `)`
Choice	⇒	*NetName ChoiceCombinator NetName*
	|	*ChoiceCombinator NetName NetName* ⌈*NetName* ⌉+
Split	⇒	*NetName SplitCombinator Range*
Range	⇒	`(` *Tag* ⌈ `:` *Tag* ⌉ `)`
SerialCombinator	⇒	`..`
StarCombinator	⇒	`*` | `**`
ChoiceCombinator	⇒	`|` | `||`
SplitCombinator	⇒	`!` | `!!`

Figure 2.19: Grammar of S-Net$_{\text{final}}$
(continued from previous page)

Chapter 3

Code Generation

3.1 Overview

The code generation is performed after completing the postprocessing. To exemplify the process, one entity from the running example (see Fig. 2.18) is translated for each combinator. The generated code contains one function for each entity in the final S-Net code. The combinators and components are provided by a library, i.e. the compiler produces calls to library functions. All entity specific information, such as type signatures (see section 3.2), are created by the compiler and then passed to the library function as parameters. To seperate the S-Net namespace, all compiled functions are prefixed with `SNet__` by the compiler.

3.2 Type representation

The symbolic names of fields, tags and binding tags in S-Net code are represented by integers in the generated code. The runtime library provides functions to create type representations. The following shows their signatures and expected parameters. A concrete example for an encoding of the type {{A,B,<T>},{D,<#BT>}} exemplifies the usage.

Library Function

```
snet_vector_t *SNetTencCreateVector( int A, ...);
```

`A`	number of entries
`...`	entries of the vector

```
snet_variantencoding_t *SnetTencVariant( snet_vector_t *A,
                                         snet_vector_t *B,
                                         snet_vector_t *C);
```

A	vector containing field names
B	vector containing tag names
C	vector containing binding tag names

```
snet_typeencoding_t *SNetTencTypeEncode( int A, ...);
```

A	number of variants
...	variants of the type

Code example

```
#define  A  1
#define  B  2
#define  D  3
#define  T  4
#define  BT 5
snet_typeencoding_t *example;

example = SNetTencTypeEncode( 2,
            SNetTencVariantEncode(
              SNetTencCreateVector( 2, A, B),
              SNetTencCreateVector( 1, T),
              SNetTencCreateVector( 0)),
            SNetTencVariantEncode(
              SNetTencCreateVector( 1, D),
              SNetTencCreateVector( 0),
              SNetTencCreateVector( 1, BT)));
```

3.3 Header File

A header file that contains the mapping of tag-, field- and binding tag names to integers is created during the code generation process (see Fig.1.3). The file also contains external declarations of functions for external networks and their fields and tags. External names of fields, tags and binding tags are prefixed to distinguish them from local names. For the running example (Fig. 2.18), the file contains the following.

```
#define   A        1
#define   B        2
#define   C        3
#define   D        4
#define   P        5
#define   Q        6
#define   T        7
#define   Y        8
#define   Z        9
#define   tag__A  10
#define   tag__B  11
#define   tag__T  12

external SNet__tag( snet_buffer_t *inbuf);
```

3.4 Box

S-Net Code Example

```
box compA( {A} -> {P});
```

Generated Code

```
void SNet__compA( snet_handle_t *hnd) {
  snet_record_t *rec;
  void *field_A;
  int tag_Y;

  rec = SNetHndGetRecord( hnd);

  field_A = SNetRecTakeField( rec, A);
  tag_Y = SNetRecTakeTag( rec, Y);

  compA( hnd, field_A, tag_Y);
}
```

3.5 Box Wrapper

S-Net Code Example

```
net compABC___SL( {A} -> {P})
connect compA;
```

Library Function

```
snet_buffer_t *SNetBox( snet_buffer_t *A, snet_buffer_t *B,
                        snet_typeencoding_t *C)
```

- `A` buffer for incoming records
- `B` wrapper function
- `C` type of output

Generated Code

```
snet_buffer_t *SNet__compABC___SL( snet_buffer_t *in_buf) {
  snet_buffer_t *out_buf;
  snet_typeencoding *out_type;

  out_type = SNetTencTypeEncode(
               SNetTencVariantEncode(
                 SNetTencCreateVector( 1, P),
                 SNetTencCreateVector( 0),
                 SNetTencCreateVector( 0)));
```

```
  outbuf = SNetBox( inbuf, &SNet__compA, out_type);

  return( out_buf);
}
```

3.6 Serial Combinator

S-Net Code Example

```
net tag__in ( {A,B} -> {tag::A, tag::B, <tag::T>}
connect tag__translate_in .. tag__tag;
```

Library Function

```
snet_buffer_t *SNetSerial( snet_buffer_t *A, snet_buffer_t *B,
                           snet_buffer_t *C)
```

A	buffer for incoming records
B	component to be connected
C	component to be connected

Generated Code

```
snet_buffer_t *SNet__tag__in( snet_buffer_t *in_buf) {
  snet_buffer_t *out_buf;

  outbuf = SNetSerial( in_buf, &SNet__tag__translate_in, &SNet__tag__tag);

  return( out_buf);
}
```

3.7 Parallel Combinator

S-Net Code Example

```
net example__compute__compAB_compABC ( {A} -> {P} | {B} -> {Q})
connect example__compute__compAB__compABC___SL ||
        example__compute__compAB__compABC___SR__SL;
```

Library Function

```
snet_buffer_t *SNetParallel( snet_buffer_t *A, snet_buffer_t *B ,
                             snet_buffer_t *C, snet_typeencoding_t *D)
```

- `A` buffer for incoming records
- `B` component to be connected
- `C` component to be connected
- `D` type containing two variants, first variant encodes input type of component A, second variant encodes input type of component B

Generated Code

```
snet_buffer_t *SNet__example__compute__compAB_compABC( buffer_t *in_buf) {
  snet_buffer_t *out_buf;

  out_buf = SNetParallel( in_buf, &SNet__example__compute__compAB__compABC___SL,
                                  &SNet__example__compute__compAB__compABC___SR__SL,
                                  SNetTencTypeEncode( 2,
                                    SNetTencVariantEncode(
                                      SNetTencCreateVector( 1, A),
                                      SNetTencCreateVector( 0),
                                      SNetTencCreateVector( 0)),
                                    SNetTencVariantEncode(
                                      SNetTencCreateVector( 1, B),
                                      SNetTencCreateVector( 0),
                                      SNetTencCreateVector( 0))));

  return( out_buf);
}
```

3.8 Split Combinator

S-Net Code example

```
net example__compute ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q})
connect example__compute___IS !! (<T>);
```

Library Function

```
SnetSplit( snet_buffer_t *A, snet_buffer_t *B, int C, int D)
```

- `A` buffer for incoming records
- `B` operand component
- `C` name of tag containing lower value
- `D` name of tag containing upper value

Generated Code

```
snet_buffer_t *SNet__example__compute( buffer_t *in_buf) {
  snet_buffer_t *out_buf;
```

```
  out_buf = SNetSplit( in_buf, &SNet__example_compute___IS, T, T);

  return( out_buf);
}
```

3.9 Star Combinator

S-Net Code Example

```
net example ({A,B} -> {Y,Z})
connect example___ST ** ({Y,Z});
```

Library Function

```
SNetStar( snet_buffer_t *A, snet_buffer_t *B, snet_buffer_t *C,
          snet_typeencoding_t *D)
SNetStarInner( snet_buffer_t *A, snet_buffer_t *B, snet_buffer_t *C,
               snet_typeencoding_t *D)
```

- `A` buffer for incoming records
- `B` operand component
- `C` function calling `SNetStarInner` for this component
- `D` exit patterns, one pattern per variant

Generated Code

```
snet_buffer_t *SNet_STAR__INNER__example___ST( snet_buffer_t *in_buf) {
  snet_buffer_t *out_buf;

  out_buf = SNetStarInner( in_buf, &SNet__example___ST,
                           &SNet_STAR__INNER__example,
                           SNetTencTypeEncode( 1,
                             SNetTencVariantEncode(
                               SNetTencCreateVector( 2, Y, Z),
                               SNetTencCreateVector( 0),
                               SNetTencCreateVector( 0))));

  return( out_buf);
}

snet_buffer_t *SNet__example___ST( snet_buffer_t *in_buf) {
  snet_buffer_t *out_buf;

  out_buf = SNetStar( in_buf, &SNet__example___ST,
                      &SNet_STAR__INNER__example,
                      SNetTencTypeEncode( 1,
                        SNetTencVariantEncode(
                          SNetTencCreateVector( 2, Y, Z),
                          SNetTencCreateVector( 0),
                          SNetTencCreateVector( 0))));
```

```
  return( out_buf);
}
```

3.10 Syncro Cell

S-Net Code

```
net example__compute__syncPQ___ST ({P} -> {P} | {P,Q}, {Q} -> {Q} | {P,Q})
connect [| {P}, {Q}|];
```

Library Function

```
SNetSync( snet_buffer_t *A, snet_typeencoding_t *B, snet_typeencoding_t *C)
```

- `A` buffer for incoming records
- `B` output type in case of synchronisation
- `C` patterns to be merged, one pattern per variant

Generated Code

```
snet_buffer_t *SNet__example__compute___syncPQ___ST( buffer_t *in_buf) {
snet_buffer_t *out_buf;

  out_buf = SNetSync( in_buf, SNetTencTypeEncode( 1,
                                SNetTencVariantEncode(
                                  SNetCreateVector( 2, P, Q),
                                  SNetCreateVector( 0),
                                  SNetCreateVector( 0))),
                              SNetTencTypeEncode( 2,
                                SNetTencVariantEncode(
                                  SNetTencCreateVector( 1, P),
                                  SNetTencCreateVector( 0),
                                  SNetTencCreateVector( 0)),
                                SNetTencVariantEncode(
                                  SNetTencCreateVector( 1, Q),
                                  SNetTencCreateVector( 0),
                                  SNetTencCreateVector( 0))));

  return( out_buf);
}
```

3.11 Filter

S-Net Code Example

```
net example__split ({A,B,<T>} -> {A,<T>} | {B,<T>})
connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];
```

Library Function

```
SNetFilter( snet_buffer_t *A, snet_typeencoding_t *B,
            snet_typeencoding_t *C, ...)
```

A	buffer for incoming records
B	type of incoming records
C	output type
...	one instruction set for each variant of the output type, all available instructions are listed in the table below

Generated Code

```
snet_buffer_t *SNet__example__split( snet_buffer_t *in_buf) {
snet_buffer_t *out_buf;

  out_buf = SNetFilter( in_buf, SNetTencTypeEncode( 1,
                                  SNetTencVariantEncode(
                                    SNetTencCreateVector( 2, A, B),
                                    SNetTencCreateVector( 1, T),
                                    SNetTencCreateVector( 0))),
                                SNetTencTypeEncode( 2,
                                  SNetTencVariantEncode(
                                    SNetTencCreateVector( 1, A),
                                    SNetTencCreateVector( 1, T),
                                    SNetTencCreateVector( 0)),
                                  SNetTencVariantEncode(
                                    SNetTencCreateVector( 1, B),
                                    SNetTencCreateVector( 1, T),
                                    SNetTencCreateVector( 0))),
                                SNetCreateFilterInstructionSet( 2,
                                  SNetCreateFilterInstruction( FLT_use_field, A),
                                  SNetCreateFilterInstruction( FLT_use_tag, T)),
                                SNetCreateFilterInstructionSet( 2,
                                  SNetCreateFilterInstruction( FLT_use_field, B),
                                  SNetCreateFilterInstruction( FLT_use_tag, T)));

  return( out_buf);
}
```

Filter Instructuions

opcode	parameter 1	parameter 2
FLT_strip_tag	tag name	—
FLT_strip_field	field name	—
FLT_add_tag	tag name	—
FLT_set_tag	tag name	integer value
FLT_rename_tag	old tag name	new tag name
FLT_rename_field	old field name	new field name
FLT_copy_field	name of source field	name of destination field
FLT_use_tag	name of tag	—
FLT_use_field	name of field, note	—
	that the field is *not* copied	

Chapter 4

Deployment

Chapter 5

Runtime System

Bibliography

[1] Shafarenko, A., Scholz, S., Grelck, C.: Streaming networks for coordinating data-parallel programs. In Virbitskaite, I., Voronkov, A., eds.: Perspectives of System Informatics, 6th International Andrei Ershov Memorial Conference (PSI'06), Novosibirsk, Russia. Volume 4378 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Heidelberg, New York (2006) 441–445

[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] Institute of Electrical and Electronic Engineers, Inc.: Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA (1995) also ISO/IEC 9945-1:1990b.

[4] Jesshope, C.: μtc — an intermediate language for programming chip multiprocessors. In: Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference (ACSAC'06), Shanghai, China. (2006)