Implementing S-Net

A Typed Stream Processing Language

— Part I —

Compilation, Code Generation and Deployment

— DRAFT —

Version 0.4

December 17, 2007

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.

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 com-

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

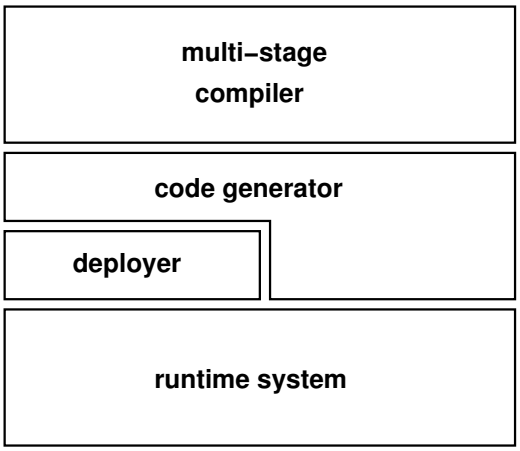

Figure 1.1: System architecture of S-Net implementation

pilation 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$_{\text{core}}$, 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.

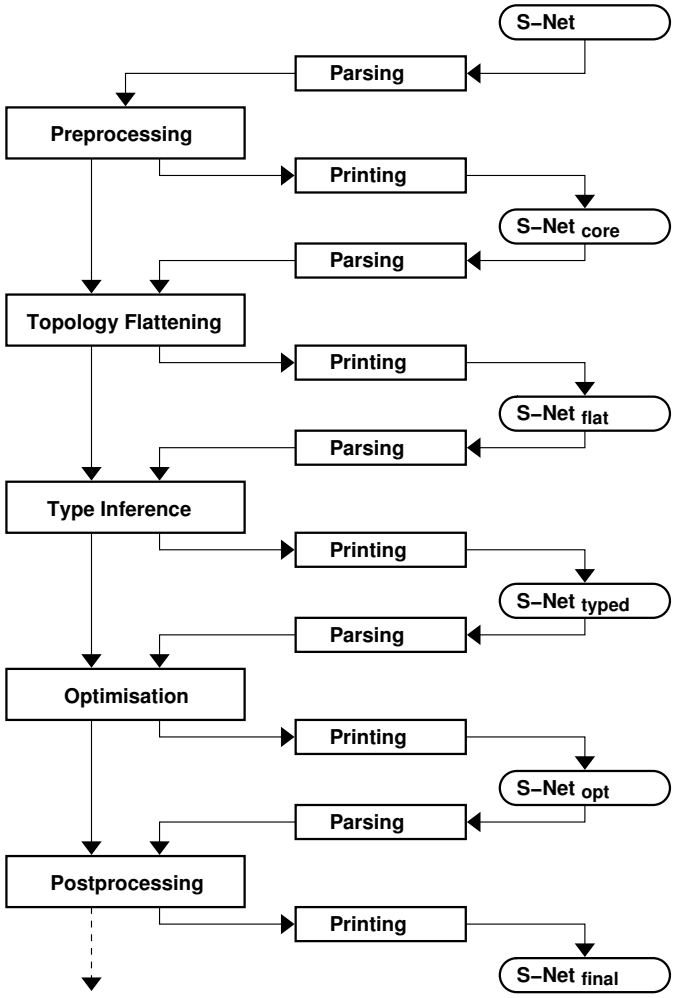

Figure 1.2: Architecture of S-Net compiler

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.

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.

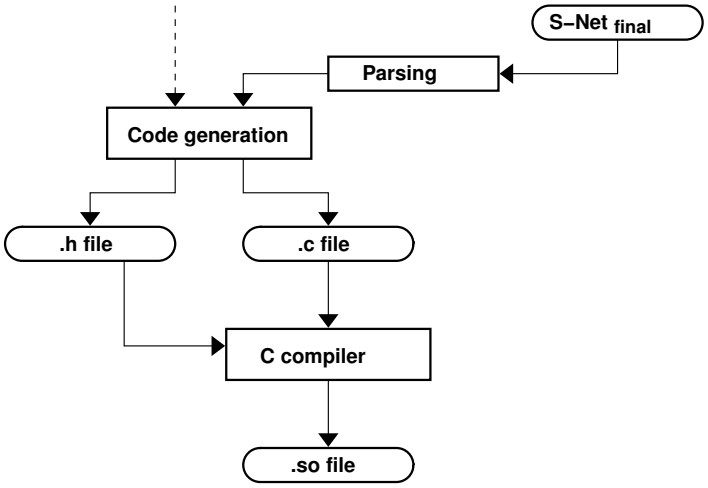

Figure 1.3: Architecture of S-Net code generator

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.

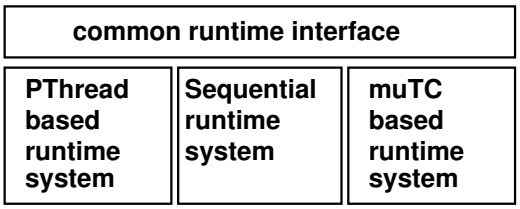

Figure 1.4: Architecture of S-Net runtime system

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 A = {A};
typesig A2P = A -> {P};
typesig compAB_t = A2P, {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.

We use user-defined type and type signature definitions towards the begin of the example to illustrate this concept, as well. However, the artificial nature of the example restricts the insight mostly to technical aspects of using type and type signature definitions in the code and their resolution by the compiler. In realistic SNets, they allow us to facilitate dealing with complex types and type signatures.

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.

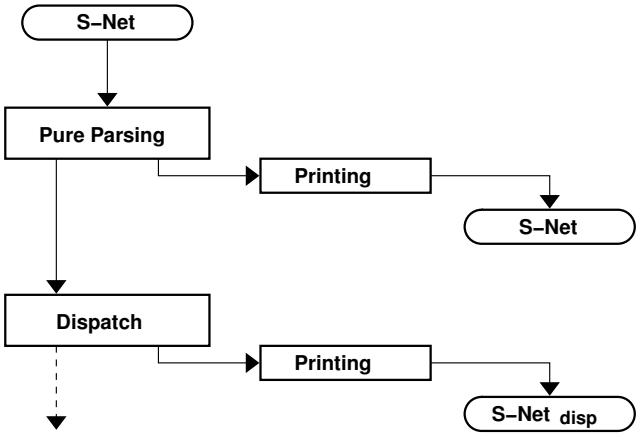

Figure 2.1: Architecture of parser

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.

$$
\begin{array}{lcl}
\mathit{SNet}_{disp} & \Rightarrow & [\,\mathit{Declaration}\,]^*\ [\,\mathit{Definition}\,]^* \\
\mathit{Declaration} & \Rightarrow & \textbf{net}\ \ \mathit{NetName}\ \ \texttt{;} \\
\mathit{TagExpr}_{disp} & \Rightarrow & \mathit{TagName} \\
 & | & \mathit{IntegerConst} \\
 & | & (\ \mathit{TagExpr}\ ) \\
 & | & (\ \mathit{UnaryOperator}\ \mathit{TagExpr}\ ) \\
 & | & (\ \mathit{TagExpr}\ \mathit{BinaryOperator}\ \mathit{TagExpr}\ ) \\
 & | & (\ \mathit{TagExpr}\ \texttt{?}\ \mathit{TagExpr}\ \texttt{:}\ \mathit{TagExpr}\ ) \\
\mathit{Serial}_{disp} & \Rightarrow & (\ \mathit{SNetExpr}\ \mathit{SerialCombinator}\ \mathit{SNetExpr}\ ) \\
\mathit{Star}_{disp} & \Rightarrow & (\ \mathit{SNetExpr}\ \mathit{StarCombinator}\ \mathit{Terminator}\ ) \\
\mathit{Choice}_{disp} & \Rightarrow & (\ \mathit{SNetExpr}\ \mathit{ChoiceCombinator}\ \mathit{SNetExpr}\ ) \\
\mathit{Split}_{disp} & \Rightarrow & (\ \mathit{SNetExpr}\ \mathit{SplitCombinator}\ \mathit{Range}\ )
\end{array}
$$

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 and operator associativities and priorities by the parser. More precisely, in S-Net$_{\text{disp}}$ all (sub-)expressions both of the expression language used in filter boxes and guarded patterns as well as of the connect expressions that define the network topology are 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. 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.

```
//! snet disp

net tag;

type A = {A};
typesig A2P = A -> {P};
typesig compAB_t = A2P, {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

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.

2.2 Preprocessing

The preprocessor compiles S-Net$_{\text{disp}}$ code into the intermediate language representation 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

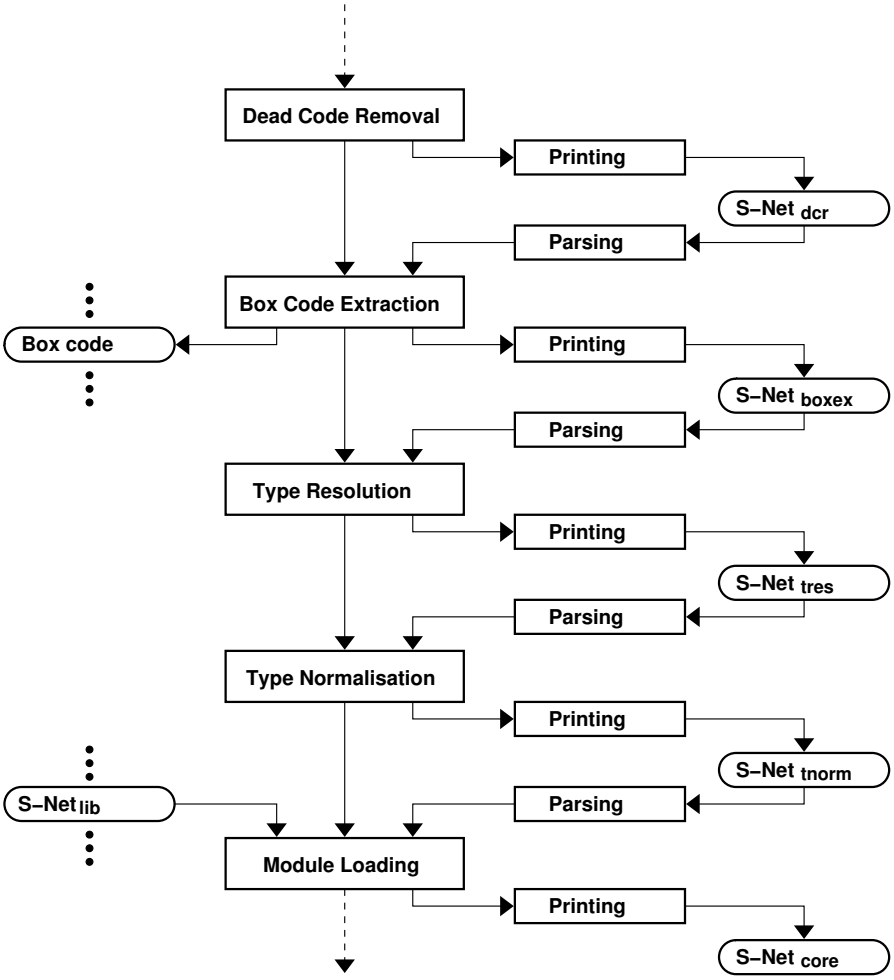

Figure 2.4: Architecture of preprocessor

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$_{\mathrm{dcr}}$ 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.

$Declaration_{core}$ ⇒ **net** *NetName* (*TypeSignature*) ;

$Definition_{core}$ ⇒ *BoxDef* | *NetDef*

$BoxDef_{core}$ ⇒ **box** *BoxName* (*BoxSignature*) ;

$TypeMapping_{core}$ ⇒ *RecordType* -> *Type*

$Type_{core}$ ⇒ *RecordType* [| *RecordType*]*

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

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 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, 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

```
//! 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

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.

$$
\begin{array}{lcl}
\mathit{SNetExpr}_{flat} & \Rightarrow & \mathit{BoxName} \\
 & | & \mathit{NetName} \\
 & | & \mathit{Sync} \\
 & | & \mathit{Filter} \\
 & | & \mathit{Combination} \\
 & & \\
\mathit{Serial}_{flat} & \Rightarrow & \mathit{NetName\ SerialCombinator\ NetName} \\
 & & \\
\mathit{Star}_{flat} & \Rightarrow & \mathit{NetName\ StarCombinator\ Terminator} \\
 & & \\
\mathit{Choice}_{flat} & \Rightarrow & \mathit{NetName\ ChoiceCombinator\ NetName} \\
 & & \\
\mathit{Split}_{flat} & \Rightarrow & \mathit{NetName\ SplitCombinator\ Range}
\end{array}
$$

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

```
//! 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)

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.

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.

```
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)

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

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

$NetDef_{typed}$ ⇒ **net** *NetName NetTypes* [*NetBody*] *Connect*

NetTypes ⇒ (*TypeSignature*) [(*NetSignature*)]

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

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.

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

```
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)

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 reasons 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.

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

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 multi-ary 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

```
//! 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)

runtime performance issues.

```
    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)

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 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.

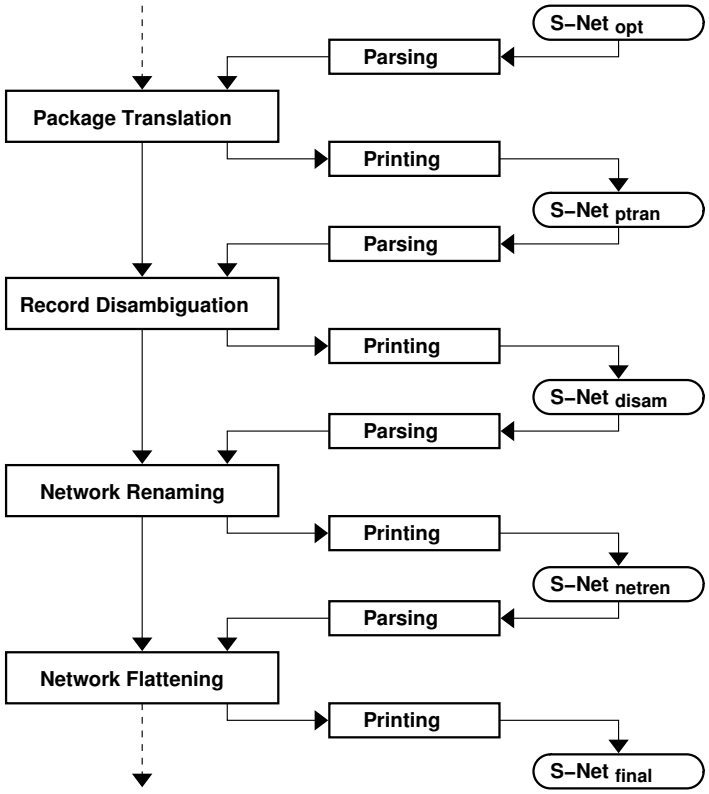

Figure 2.14: Architecture of postprocessor

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

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

```
//! 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)

```
.. 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$_{\text{ptran}}$.

$$
\begin{array}{lcl}
\mathit{Declaration}_{\mathit{ptran}} \Rightarrow & & \textbf{net}\ \mathit{Netname}\ \texttt{::}\ \mathit{NetName}\ \texttt{(}\ \mathit{TypeSignature}\ \texttt{)}\ \ \texttt{;} \\
\\
\mathit{SNetExpr}_{\mathit{ptran}} & \Rightarrow & \mathit{BoxName} \\
& | & \lceil \mathit{Netname}\ \texttt{::}\ \rceil\ \mathit{NetName} \\
& | & \mathit{Sync} \\
& | & \mathit{Filter} \\
& | & \mathit{Translator} \\
& | & \mathit{Combination} \\
\\
\mathit{Translator} & \Rightarrow & \texttt{[}\ \mathit{BoxType}\ \texttt{->}\ \mathit{BoxType}\ \texttt{]} \\
\\
\mathit{Field} & \Rightarrow & \lceil \mathit{Netname}\ \texttt{::}\ \rceil\ \mathit{FieldName} \\
\\
\mathit{SimpleTag} & \Rightarrow & \texttt{<}\ \lceil \mathit{Netname}\ \texttt{::}\ \rceil\ \mathit{SimpleTagName}\ \texttt{>} \\
\\
\mathit{BindingTag} & \Rightarrow & \texttt{<}\ \lceil \mathit{Netname}\ \texttt{::}\ \rceil\ \mathit{BindingTagName}\ \texttt{>}
\end{array}
$$

Figure 2.16: 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 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.

```
//! 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 compABC__compA compA ((A) -> (P));
  box compABC__compB compB ((B) -> (Q));
  box compABC__compC 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 example__examine examine ((P,Q) -> (A,B) | (Y,Z));
```

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

The third postprocessing phase consistently renames all networks and boxes 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 and box name is prefixed with the names of all networks in whose definitions it is embedded. The use of double underscores, which is ruled out in proper S-Net, to separate the various prefixes from each other and from the original name prevents name clashes.

Note that we do rename boxes just as networks, although boxes with the same name refer to

```
  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 example__compute__compAB__compABC__compA compA ((A) -> (P));
        box example__compute__compAB__compABC__compB 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)

the same box language implementation and, hence, should be identifiable. However, different box definitions on the level of S-Net bearing the same name may well be associated with different meta data, which mau affect their further treatment. On the one hand we need to distinguish them properly, but on the other hand we must also keep the original name to maintain the necessary link to some box language implementation. Therefore, in S-Net$_{\text{final}}$ boxes effectively have two

names: the first one is systematically renamed while the second one is the original name.

With respect to multiple box definitions bearing the same name, our running example is not representative as in our case the multiple definitions of equally named boxes stem from a specialisation performed during the optimisation phase. In general, however, equally named boxes may deliberately be put into different networks by the programmer to distinguish them with respect to meta data.

```
//! 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 compABC__compA compA ((A) -> (P));
box compABC__compB compB ((B) -> (Q));
box compABC__compC 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 example__examine examine ((P,Q) -> (A,B) | (Y,Z));

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

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

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 facilitates the realisation and maintenance of both individual phases. Fig. 2.18 shows the final S-Net representation of our running example. This concludes the

```
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)

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$_{\text{final}}$.

SNet	⇒	*⌈Declaration ⌉* ⌈Definition ⌉**
Declaration	⇒	**net** *Netname* `::` *NetName* `(` *TypeSignature* `)` `;`
Definition	⇒	*BoxDef* | *NetDef*
BoxDef	⇒	**box** *BoxName 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* `::` *⌉ SimpleTagName* `>`
BindingTag	⇒	`<` *⌈Netname* `::` *⌉ BindingTagName* `>`
BindingTagName	⇒	`#` *SimpleTagName*
Connect	⇒	**connect** *SNetExpr* `;`
SNetExpr	⇒	*BoxName*
	|	*⌈Netname* `::` *⌉ NetName*
	|	*Sync*
	|	*Filter*
	|	*Translator*
	|	*Combination*

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

Filter ⇒ `[` *Pattern* ⟦*GuardedAction* ⟧* `]`
| `[ ]`

Pattern ⇒ `{` ⟦*RecordEntry* ⟦ `,` *RecordEntry* ⟧* ⟧ `}`

GuardedAction ⇒ ⟦ **if** `<` *TagExpr* `>` ⟧ `->` ⟦*Action* ⟧

Action ⇒ *RecordOutput* ⟦ `;` *RecordOutput* ⟧*

RecordOutput ⇒ `{` ⟦*OutputField* ⟦ `,` *OutputField* ⟧* ⟧ `}`

OutputField ⇒ *FieldName* ⟦ `=` *FieldName* ⟧
| `<` *TagName* ⟦ `=` *TagExpr* `>` ⟧

TagName ⇒ *SimpleTagName* | *BindingTagName*

TagExpr ⇒ *TagName*
| *IntegerConst*
| `(` *TagExpr* `)`
| `(` *UnaryOperator TagExpr* `)`
| `(` *TagExpr BinaryOperator TagExpr* `)`
| `(` *TagExpr* `?` *TagExpr* `:` *TagExpr* `)`

UnaryOperator ⇒ `!` | `abs`

BinaryOperator ⇒ *ArithmeticOperator* | *ComparisonOperator* | *RelationalOperator* | *LogicalOperator*

ArithmeticOperator ⇒ `*` | `/` | `%` | `+` | `-`

ComparisonOperator ⇒ `min` | `max`

RelationalOperator ⇒ `==` | `!=` | `<` | `<=` | `>` | `>=`

LogicalOperator ⇒ `&&` | `||`

Sync ⇒ `[|` *GuardedPattern* ⟦ `,` *GuardedPattern* ⟧+ `|]`

GuardedPattern ⇒ *Pattern* ⟦ **if** `<` *TagExpr* `>` ⟧

Translator ⇒ `[` *BoxType* `->` *BoxType* `]`

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

Combination	⇒	*Serial* | *Star* | *Choice* | *Split*
Serial	⇒	*NetName SerialCombinator NetName*
Star	⇒	*NetName StarCombinator Terminator*
Terminator	⇒	*GuardedPattern* [, *GuardedPattern*]*
Choice	⇒	*NetName ChoiceCombinator NetName*
	|	*ChoiceCombinator NetName NetName* [*NetName*]+
Split	⇒	*NetName SplitCombinator Range*
Range	⇒	*Tag* [: *Tag*]
SerialCombinator	⇒	..
StarCombinator	⇒	* | **
ChoiceCombinator	⇒	| | ||
SplitCombinator	⇒	! | !!

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

Chapter 3

Code Generation

3.1 Overview

This chapter outlines the code generation phase. 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 and each function is constructed in the same way. The combinators and components are provided by the runtime library, i.e. the compiler produces calls to library functions. The signature of a generated function is always the same; it expects a buffer as parameter and returns a buffer after completion. All information that are specific for a concrete application of a component, such as type signatures (see section 3.2) and names of operands, 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

```
snet_typeencoding_list_t *SNetTencCreateTypeEncodingList( int A, ...);
```

`A`	number of type encodings
`...`	type encodings

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 Expressions

The runtime system supports guards and arithmetic expressions as they are used by filters, synchronisation boxes and star combinators. To encode guards and arithmetic operations the runtime system provides a simple expression language.

Runtime System	Description	S-Net expression
`SNetEconsti( 47)`	integer constant	47
`SNetEconstb( true)`	boolean constant	`true`
`SNetEtag( a)`	value of tag a	`<a>`
`SNetEbtag( a)`	value of binding tag a	`<#a>`
`SNetEabs( a)`	absolute value of a	abs a
`SNetEadd( a,b)`	add a and b	$a + b$
`SNetEmul( a,b)`	multiply a and b	$a \cdot b$
`SNetEsub( a,b)`	subtract b from a	$a - b$
`SNetEdiv( a,b)`	divide a by b	a/b
`SNetEmod( a,b)`	remainder of a divided by b	$a \mod b$
`SNetEmin( a,b)`	minimum of a and b	a min b
`SNetEmax( a,b)`	maximum of a and b	a max b
`SNetEeq( a,b)`	bool: a equals b	$a == b$
`SNetEne( a,b)`	bool: a not equal b	$a != b$
`SNetEgt( a,b)`	bool: a greater than b	$a > b$
`SNetEge( a,b)`	bool: a greater or equal b	$a >= b$
`SNetElt( a,b)`	bool: a less than b	$a < b$
`SNetEle( a,b)`	bool: a less than or equal b	$a <= b$
`SNetEand( a,b)`	bool: a and b	a && b
`SNetEor( a,b)`	bool: a or b	$a \mid\mid b$
`SNetEnot( a)`	bool: not a	$!a$
`SNetEcond( a,b,c)`	if a then b else c	$a?b : c$

Library Functions

The following shows the signatures of provided functions. Due to the fact that most signatures differ from each other solely by their names, only selected functions are shown.

```
snet_expr_t *SNetEconsti( int val);

snet_expr_t *SNetEtag( int tag_name);

snet_expr_t *SNetEnot( snet_expr_t *A);

snet_expr_t *SNetEadd( snet_expr_t *A,
                       snet_expr_t *B);

snet_expr_t *SNetEcond( snet_expr_t *A,
                        snet_expr_t *B,
                        snet_expr_t *C);

snet_expr_list_t *SNetEcreateList( int num, ...);
```

Code Examples

The encoding of the expressions `(47+23) / 25` and `<T> + 1` is shown in the following example source code. The lower part of the code shows how to create a list of these expressions.

```
snet_expr_t *my_expr_a, *my_expr_b;
snet_expr_list_t *my_expr_list;

my_expr_a = SNetEdiv(
              SNetEadd( SNetEconsti( 47), SNetEconsti( 23)),
```

```
                SNetEconsti( 25));

  my_expr_b = SNetEadd( SNetEtag( T), SNetEconsti( 1));

  my_expr_list = SNetEcreateList( 2, my_expr_a, my_expr_b);
```

3.4 Header File

As a part of the code generation process (see Fig.1.3) the S-Net compiler creates a header file that contains a prototype declaration for the C function that is to be compiled from the outermost (i.e. exported) S-Net network along with C preprocessor instructions that implement a mapping between S-Net labels and integer numbers. For the running example (see Fig. 2.18), the corresponding header file is shown in Fig. 3.1.

```
#ifndef _SNET__example_h_
#define _SNET__example_h_

#include "snetentities.h"

#define F__example__A  1
#define F__example__B  2
#define F__example__C  3
#define F__example__P  4
#define F__example__Q  5
#define F__example__T  6
#define F__example__Y  7
#define F__example__Z  8
#define T__example__A  9
#define T__example__B 10
#define T__example__T 11

extern snet_buffer_t *SNet__example( snet_buffer_t *in_buf);

#endif
```

Figure 3.1: Header file generated for running example

Lines 1, 2 and 16 are just aimed at avoiding the repeated evaluation of the header file, and line 3 includes the standard header file from the S-Net distribution, which is needed for the definition of the `snet_buffer_t` type used in line 15.

Lines 4–14 implement the mapping from textual labels to integer numbers. In order to avoid name clashes we prefix each label with a single character indicating the type of label: “F” for field, “T” for tag and “B” for binding tag. Separated by a double underscore we further prefix each label by the module name (i.e. the name of the exported network). This allows us to distinguish between labels introduced in separately specified S-Net definitions. Finally, after another double underscore we have the original label. Note that both the module name as well as the label are case-sensitive.

3.5 Box

The generated function for a box is the only one not following the general scheme. It calls the computation function that is implemented in an arbitrary language obeying the box language

contract. A record is embedded in the opaque data object passed to this function. From that record, all needed fields, tags and binding tags are extracted and then passed to the computation function. To extract the data, library functions are provided.

S-Net Code Example

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

Library Function

```
snet_record_t *SNetHndGetRecord( snet_handle_t *A);
```

A	handle data object which was passed to the function

```
void *SNetRecTakeField( snet_record_t *A, int B);
int SNetRecTakeTag( snet_record_t *A, int B);
int SNetRecTakeBTag( snet_record_t *A, int B);
```

A	record containing data
B	name of field, tag, binding tag

Generated Code

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

  rec = SNetHndGetRecord( hnd);

  field_A = SNetRecTakeField( rec, A);

  compA( hnd, field_A);
}
```

3.6 Box Wrapper

S-Net Code Example

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

Library Function

```
snet_buffer_t *SNetBox( snet_buffer_t *A, void (*B)( snet_handle_t*),
                        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( 1,
               SNetTencVariantEncode(
                 SNetTencCreateVector( 1, P),
                 SNetTencCreateVector( 0),
                 SNetTencCreateVector( 0)));
  outbuf = SNetBox( in_buf, &SNet__compA, out_type);

  return( out_buf);
}
```

3.7 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*),
                           snet_buffer_t* (*C)( snet_buffer_t*))
```

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);
}
```

3.8 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_typeencoding_t *B,
                             ...)
```

- A buffer for incoming records
- B type containing variants, one per component

Generated Code

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

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

  return( out_buf);
}
```

3.9 Split Combinator

S-Net Code example

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

Library Function

```
snet_buffer_t *SnetSplit( snet_buffer_t *A,
                          snet_buffer_t* (*B)(*snet_buffer_t*),
                          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);
}
```

3.10 Star Combinator

The star combinator requires two functions to be compiled. One function initialises all internal data structures and is called only once, whereas the second function (calling `SNetStarIncarnate`) is called for each dynamically created incarnation of the component. The function called for incarnations is prefixed by `__STAR_INCARNATE_`. The guards of exit patterns are encoded as an expression list. If a pattern is not protected by a guard, a `NULL` pointer is allowed as expression. If the star combinator does not use any guards, `NULL` may be passed as expression list.

S-Net Code Examples

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

net guard_example ({A,B} -> {Y,Z,<T>})
connect guard_example___ST *({Y,Z,<T>} if <T==42>);
```

Please note that the code of the incarnate function of the lower example is not shown in the *Generated Code* section.

Library Function

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

snet_buffer_t *SNetStarIncarnate( snet_buffer_t *A,
                                  snet_typeencoding_t *B,
                                  snet_expr_list_t *C,
                                  snet_buffer_t* (*D)(snet_buffer_t*),
                                  snet_buffer_t* (*E)(snet_buffer_t*))
```

- `A` buffer for incoming records
- `B` exit patterns, one pattern per variant
- `C` list of guard expressions, one expression per pattern
- `D` operand component
- `E` function calling `SNetStarIncarnate` for this component

Generated Code

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

  out_buf = SNetStarIncarnate( in_buf,
              SNetTencTypeEncode( 1,
                SNetTencVariantEncode(
                  SNetTencCreateVector( 2, Y, Z),
```

```
                  SNetTencCreateVector( 0),
                  SNetTencCreateVector( 0))),
              NULL,
              &SNet__example___ST,
              &SNet____STAR_INCARNATE_example);

  return( out_buf);
}

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

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

  return( out_buf);
}

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

  out_buf = SNetStar( in_buf,
              SNetTencTypeEncode( 1,
                SNetTencVariantEncode(
                  SNetTencCreateVector( 2, Y, Z),
                  SNetTencCreateVector( 1, T),
                  SNetTencCreateVector( 0))),
              SNetEcreateList( 1, SNetEeq( SNetEtag( T), SNetEconsti( 42)),
              &SNet__example___ST,
              &SNet____STAR_INCARNATE_guard_example);

  return( out_buf);
}
```

3.11 Syncro Cell

The guards of patterns are encoded as an expression list. If a pattern is not protected by a guard, a `NULL` pointer is allowed as expression. If the synchro cell does not use any guards, `NULL` may be passed as expression list.

S-Net Code

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

net guard_example__compute__syncPQ___ST ( {P,<T>} -> {P,<T>} | {P,Q,<T>},
                                          {Q} -> {Q} | {P,Q,<T>})
connect [| {P,<T>} if <T == 1>, {Q}|];
```

Library Function

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

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

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))),
                              NULL);

  return( out_buf);
}

snet_buffer_t *SNet__guard_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( 1, T),
                                  SNetCreateVector( 0))),
                              SNetTencTypeEncode( 2,
                                SNetTencVariantEncode(
                                  SNetTencCreateVector( 1, P),
                                  SNetTencCreateVector( 1, T),
                                  SNetTencCreateVector( 0)),
                                SNetTencVariantEncode(
                                  SNetTencCreateVector( 1, Q),
                                  SNetTencCreateVector( 0),
                                  SNetTencCreateVector( 0))),
                              SNetEcreateList( 2,
                                SNetEeq( SNetEtag( T), SNetEconsti( 1)),
                                NULL));

  return( out_buf);
}
```

3.12 Filter

The output type of the filter is derived from filter instructions. The filter instructions describe how an inbound record is transformed into one or more outbound record(s). A field or tag that is present in the inbound type of the filter will be taken over to an outbound record if and only if it is explicitly triggered by a filter instruction. A filter that only removes fields or tags from inbound records requires a `record_create` instruction. If no records are produced (consumer) `NULL` is to be passed instead of an instruction. Each action of the filter is expressed as list of instruction sets. Each set of the list results in one outbound record. The filter allows guard expressions for each action. These guards are passed as a list of expressions, where entry i of the list protects action i of the filter. If an action is not protected by a guard, a `NULL` pointer is allowed as expression. If the filter does not use any guards (single case filter), `NULL` may be passed as expression list.

Library Function

```
snet_filter_instruction_t
*SNetCreateFilterInstruction( snet_filter_opcode_t A, ...)
```

`A`	opcode defining the action to be perfomed
`...`	parameter(s) needed for the action (see below)

```
snet_filter_instruction_set_t
*SNetCreateFilterInstructionSet( int A, ...)
```

`A`	number of instructions for this set
`...`	filter instructions

```
snet_filter_instruction_set_list_t
*SNetCreateFilterInstructionSetList( int A, ...)
```

`A`	number of instruction sets for this list
`...`	instruction sets

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

`A`	buffer for incoming records
`B`	type of incoming records
`C`	list of guard expressions, entry i of this list corresponds to entry i of the type list
`...`	one instruction set list for each entry of the expression list; 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))),
                        NULL,
                        SNetCreateFilterInstructionSetList( 2,
```

```
                            SNetCreateFilterInstructionSet( 2,
                              SNetCreateFilterInstruction( snet_field, A, A),
                              SNetCreateFilterInstruction( snet_tag, SNetEtag( T))),
                            SNetCreateFilterInstructionSet( 2,
                              SNetCreateFilterInstruction( snet_field, B, B),
                              SNetCreateFilterInstruction( snet_tag, SNetEtag( T)))));

    return( out_buf);
  }

snet_buffer_t *SNet__guard_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))),
                          SNetEcreateList( 2,
                            SNetEeq( SNetEtag( T), SNetEconsti( 42)),
                            NULL),
                          SNetCreateFilterInstructionSetList( 1,
                            SNetCreateFilterInstructionSet( 2,
                              SNetCreateFilterInstruction( snet_field, A, A),
                              SNetCreateFilterInstruction( snet_field, B, B))),
                          SNetCreateFilterInstructionSetList( 2,
                            SNetCreateFilterInstructionSet( 2,
                              SNetCreateFilterInstruction( snet_field, A, A),
                              SNetCreateFilterInstruction( snet_tag, SNetEtag( T))),
                            SNetCreateFilterInstructionSet( 2,
                              SNetCreateFilterInstruction( snet_field, B, B),
                              SNetCreateFilterInstruction( snet_tag, SNetEtag( T)))));


    return( out_buf);
```

Filter Instructions

opcode	parameter 1	parameter 2
create_record	generate empty record	none
snet_field	name of field in outbound record	name of field in inbound record
snet_tag	name of tag in outbound record	expression to initialise tag
snet_btag	name of binding tag in outbound record	expression to initialise binding tag

3.13 Deterministic Combinators

The signatures of the deterministic variants of the combinators are consistent with the non-deterministic implementations. They are listed here for reference and completeness.

Deterministic Parallel Combinator

```
  snet_buffer_t *SNetParallelDet( snet_buffer_t *A,
                                  snet_typeencoding_t *B)
                                  ...)
```

- `A` buffer for incoming records
- `B` typeencoding that contains one variant per component

Deterministic Split Combinator

```
snet_buffer_t *SnetSplitDet( snet_buffer_t *A,
                             snet_buffer_t* (*B)(*snet_buffer_t*),
                             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

Deterministic Star Combinator

```
snet_buffer_t *SNetStarDet( snet_buffer_t *A,
                            snet_typeencoding_t *B,
                            snet_expr_list_t *C,
                            snet_buffer_t* (*D)(snet_buffer_t*),
                            snet_buffer_t* (*E)(snet_buffer_t*))

snet_buffer_t *SNetStarDetIncarnate( snet_buffer_t *A,
                                     snet_typeencoding_t *B,
                                     snet_expr_list_t *C,
                                     snet_buffer_t* (*D)(snet_buffer_t*),
                                     snet_buffer_t* (*E)(snet_buffer_t*))
```

- `A` buffer for incoming records
- `B` exit patterns, one pattern per variant
- `C` list of guard expressions, one expression per pattern
- `D` operand component
- `E` function calling `SNetStarDetIncarnate` for this component

Chapter 4

Deployment

Chapter 5

Runtime System

Chapter 6

Language Interfaces

6.1 Preliminaries

The following introduces the naming convention for functions and the minimal functionality an interface implementation is expected to provide.

The S-Net compiler generates calls to initialisation functions for each interface a particular SNet uses. Therefore, any interface implementation provides:

```
void IfID_init( int id);
```

Any function that the interface implementation provides must be prefixed by a unique identification string (denoted by `IfID` in the above example) to separate namespaces of different interface implementations. All interfaces need to be registered with the runtime system. This is achieved by calling

```
bool SNetGlobalRegisterInterface( int id,
                                  void (*freefun)( void*),
                                  void* (*copyfun)( void*));
```

from within the `IdID_init` function, where `*freefun` and `*copyfun` are pointers to the language specific free and copy functions[1].

Results that boxes produce need to be processed by the interface implementation before they may enter the S-Net domain. The runtime system provides the following functions to communicate back the results:

```
snet_handle_t *SNetOutRaw( snet_handle_t *hnd,
                           int if_id, int variant_num,
                           ...);

snet_handle_t *SNetOutRawArray( snet_handle_t *hnd,
                                int if_id,
                                int var_num,
                                void **fields,
                                int *tags,
                                int *btags);
```

[1] This will be extended by (de-)serialise functions in the future.

The `if_id` that is passed to the function is the same as the one assigned to the interface by the compiler (see above). The arguments following the variant number are the produced fields followed by produced tags followed by produced binding tags. The order of fields/tags/binding tags passed to `SNetOutRaw` is defined by the order in which they appear in the output type of the box in the S-Net source code. The same holds for `SNetOutRawArray`. To use the latter function fields/tags/binding tags have to be grouped into arrays first.

6.1.1 Calling a box function

S-Net requires any box function to accept an opaque handle object as its first parameter. Fields, tags and binding tags are passed to the box function in the order they appear in the input type of the box as specified in the S-Net source code.

6.2 C Interface

The `C` interface implementation provides a `C_Data` structure to encapsulate arbitrary data. The `C_Data` structure contains the data itself and annotates free and copy function to compensate for the lack of generic free and copy functions in `C`.

The `C_Data` structure is maintained by the following functions:

```
C_Data *C2SNet_cdataCreate( void *A,
                            void (*B)( void*),
                            void* (*C)( void*));

void *C2SNet_cdataGetData( C_Data *D);
void *C2SNet_cdataGetCopyFun( C_Data *D);
void *C2SNet_cdataGetFreeFun( C_Data *D);
void C2SNet_cdataDestroy( C_Data *D);
```

- `A` pointer to data
- `B` data specific free function
- `C` data specific copy function
- `D` pointer to `C_data` structure

To create a `C_Data` structure a pointer to the data to be encapsulated, a pointer to a data specific free function and a pointer to a data specific copy function are passed to `C2SNet_cdataCreate`.

The interface implementation provides the obligatory initialisation function:

```
void C2SNet_init( int A);
```

- `A` integer id (assigned by compiler)

A box that utilises this `C` interface may communicate back results in two different ways. The interface implementation provides a variadic function to send back results with a single function call.

```
void C2SNet_outRaw( void *A, int B, ...);
```

- `A` pointer to opaque handle object (passed as parameter to the box function)
- `B` variant number
- `...` produced results

The arguments following the variant number are the produced fields followed by produced tags

followed by produced binding tags. Fields, tags and binding tags are in the same order as they appear in the box output type (defined in the S-Net source code). Fields are pointers to `C_Data`; tags and binding tags are of type `int`.

The interface implementation also provides functions to store results in a data object.

```
c2snet_container_t *C2SNet_containerCreate( void *A, int B);
c2snet_container_t *C2SNet_containerSetField( c2snet_container_t *C, void *D);
c2snet_container_t *C2SNet_containerSetTag( c2snet_container_t *C, int E;
c2snet_container_t *C2SNet_containerSetBTag( c2snet_container_t *C, int E);

void C2SNet_out( c2snet_container_t *c);
```

- `A` pointer to opaque handle object (passed as parameter to the box function)
- `B` variant number
- `C` pointer to container object
- `D` pointer to data
- `E` integer value of tag / binding tag

To create a container the opaque handle object and the desired variant number are passed to `C2SNet_containerCreate`. The `C2SNet_containerSetX` functions, where `X` is Field, Tag or BTag, fill the container successively in the same order as fields/tags/btags appear in the output type of the box[2].

6.2.1 Example

Still to come in this section:
Metadata to annotate interface and function names.

S-Net:

```
net simple {
        box sub( (x) -> (xx));
        box mult( (x,r) -> (rr));
} connect sub | mult;
```

Box Functions:

```
/* myfuns.c
 * ------------------------------------------
 * Implementation of free- and copy function
 */
#include <stdlib.h>
#include <myfuns.h>

void myfree( void *ptr)
{
  ...
}
```

[2]This is a very primitive implementation but may be extended in the future.

```
void *mycopy( void *ptr)
{
  ...
}
```

```
/* sub.c
 * ------------------------------------------
 * Implementation of box function for box "sub"
 */

#include <stdlib.h>
#include <sub.h>
#include <myfuns.h>

void *sub( void *hnd, C_Data *x)
{
  int *int_x;
  c2snet_container_t *c;
  C_Data *result;

  int_x = malloc( sizeof( int));

  *int_x= *(int*)C2SNet_cdataGetData( x);

  /* ---  do complex computation ---> */

  *int_x -= 1;

  /* <--- computation end --- */

  result = C2SNet_cdataCreate( int_x, &myfree, &mycopy);

  C2SNet_outRaw( hnd, 1, result);

  return( hnd);
}
```

```
/* mult.c
 * ------------------------------------------
 * Implementation of box function for box "mult"
 */

#include <stdlib.h>
#include <mult.h>
#include <myfuns.h>

void *mult( void *hnd, C_Data *x, C_Data *r)
{
  c2snet_container_t *c;
  C_Data *result;

  int int_x, int_r, *int_rr;

  c = C2SNet_containerCreate( hnd, 1);

  int_rr = malloc( sizeof( int));

  int_x = *(int*) C2SNet_cdataGetData( x);
```

```
  int_r = *(int*) C2SNet_cdataGetData( r);
  C2SNet_cdataDestroy( x);
  C2SNet_cdataDestroy( r);

  /* ---  do complex computation ---> */

  *int_rr = int_x * int_r;

  /* <--- computation end --- */

  result = C2SNet_cdataCreate( int_rr, &myfree, &mycopy);

  C2SNet_containerSetField( c, result);

  C2SNet_output( c);

  return( hnd);
}
```

6.3 SAC Interface

The SAC interface is in an intermediate state. For the time being a SAC function communicates back results via its `return` statement. This requires a wrapper around the function call to pass returned values on to a function that communicates the results back to S-NET.

The interface implementation provides the obligatory initialisation function:

```
void SAC2SNet_init( int A, int B);
```

- `A` integer id (assigned by compiler)
- `B` basetype as assigned in `cwrapper.h` by *sac4c*

The interface implementation provides a function to pass results to S-NET:

```
void SAC2SNet_outRaw( void *A, int B, ...);
```

- `A` pointer to opaque handle object (passed as parameter to the box function)
- `B` variant number
- `...` produced results

The arguments following the variant number are the produced fields followed by produced tags followed by produced binding tags. Fields, tags and binding tags are in the same order as they appear in the box output type (defined in the S-Net source code). All results are pointers to `SACarg`s.

6.3.1 Example

S-Net:

```
net simple
{
  box create( (dim) -> (x,z));
  box dec( (x) -> (y));
```

```
} connect create .. dec;
```

Wrapper Code:

```
/* create.c
 * ----------------------------------------
 * Wrapper code for box function "create"
 */

#include <create.h>
#include <cwrapper.h> /* created by sac4c */

void *create( void *hnd, void *dim)
{
  SACarg *sac_result_x, *sac_result_z;

  /* SAC function */
  simple__create1( &sac_result_x, &sac_result_z, dim);

  SAC2SNet_outRaw( hnd, 1, sac_result_x, sac_result_z);

  return( hnd);
}
```

```
/* dec.c
 * ----------------------------------------
 * Wrapper code for box function "dec"
 */

#include <dec.h>
#include <cwrapper.h> /* created by sac4c */

void *dec( void *hnd, void *x)
{
  SACarg *sac_result_y;

  /* SAC function */
  simple__dec1( &sac_result_y, x);

  SAC2SNet_outRaw( hnd, 1, sac_result_y);

  return( hnd);
}
```

SaC Code:

```
module simple;

use Numerical: all;
use Structures: all;

export all;


int[*], int[*] create( int dim)
{
```

```
        return( genarray( [dim], 17), genarray( [dim], 42));
}

int[*] dec( int[*] x)
{
        return( x - 1);
}
```

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 (2007) 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)