Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND **D**YNAMIC **V**IRTUALISATION THROUGH PERFORMANCE
ANALYSIS TO SUPPORT **C**ONCURRENCY **E**NGINEERING

Report describing VR-Net and interfaces

D11

Due date of deliverable: May 31st, 2011
Actual submission date: September xxth, 2011

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP2
Task number: WP2c

Responsible institution: HERTS
Editor & editor's address: Raimund Kirner
University of Hertfordshire, College Lane
Hatfield, AL10 9AB, United Kingdom

Version 1.0 / Last edited by Raimund Kirner / September 6th, 2011

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	15/05/2011	Raimund Kirner	HERTS	Initial version
0.2	17/05/2011	Clemens Grelck	UvA	Topology flattening and intermediate representation
0.3	18/05/2011	Frank Penczek	HERTS	Running example
0.4	20/05/2011	Alex Shafarenko	HERTS	Motivation
1.0	06/09/2011	Raimund Kirner	HERTS	Final version

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP2c	VR-Net Definition Interfaces	HERTS*, USTAN, UvA, TWENTE

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

Within the **Advance** project we extent key technologies for the development and execution of concurrent programs to deploy the optimisation potential of runtime optimisations based on statistical program information.

In this document we describe an intermediate representation of S-Net programs that annotates indispensable information to facilitate automatic aggregation of statistical properties. The technical details of this transformation process are laid out and are illustrated by means of a non-trivial example.

Contents

Chapter 1

Introduction

In the context of the **Advance** project several development lines need to interface with one another. Due to the shortness of the project term it is unrealistic to expect co-development of the new technologies by organisations that are not on the same physical site. As always in computer science the solution to consistency problems should be in terms of defining abstract interfaces for all the co-operating tools, and then those can then be taken away and used at remote sites with assured compatibility.

The specific problem with S-Net is that type information is used in more than one way in the language. On the one hand, it is used to ensure the output-to-input consistency between components, where one component is capable of producing messages of a certain structure and another potentially receives these messages and needs a static assurance that it will be able to process them. This side of the type system has a purpose internal to S-Net and not affecting any collaborative work. However, types are used in the language for routing decisions also: if a message can choose from several pathways to follow, the choice must be consistent with the message type. Routing is inexorably linked with resource management, property aggregation and virtual hardware and thus access to the routing information is required for effective collaboration.

Consequently, a technology that deals with logic around messages: the prediction of where the messages will be routed, which paths they will follow and how statically inferred information about the components might alter these paths is required for the property-aggregation tools, but importantly this information is not *immediately* available from an ordinary S-Net program. It can always be inferred by the type system, and so collaborating partners have a choice of either sharing the type-inference code, or merely annotating S-Net programs with *all* type information that can possibly be inferred. In addition to that, the S-Net program must be *flattened*, i.e. any hierarchically enveloping structures provided by the language for the purposes of expressiveness and conciseness should be substituted throughout to avoid the need to follow the program structure in an analysis tool.

These two actions, flattening and type annotation, can be done without breaking

the syntax of S-Net. Thus the aforementioned abstract interface is the language itself, in which certain severe restrictions are imposed. We call this restricted form of S-Net VR-Net.

The following sections introduce and explain the structures of VR-Net and give illustrations of VR-Net produced from some source S-Net examples.

Chapter 2

Virtual Resource Network (VR-Net)

2.1 Description of VR-Net

VR-Netencodes the flat network structure with all type information annotated. A program in VR-Net notation is still a legal S-Net program. The generally complex network structure of an S-Net network has been fully flattened into a strict component and single combinator representation. In order to achieve this, auxiliary networks are introduced as wrappers around the original network segments. At all network boundaries the type information has been inferred and annotated to encode the input and output behaviour.

The transformation process into the flat VR-Net representation is primarily driven by a transformation component, which we call *topology flatter*. The type annotations are automatically derived by a type inference system as described in [1].

2.2 Running example

We illustrate the transformation process from S-Net to VR-Net by means of the running example shown in Fig. 2.1. 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 network rather than a concrete S-Net application.

The SNet `example` in Fig. 2.1 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. 2.1, 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

```
1  type A = {A};
2  typesig A2P = A -> {P};
3  typesig compAB_t = A2P, {B} -> {Q};
4
5  net compABC (A | {C} -> {P}, {B} -> {Q}) {
6    box compA ((A) -> (P));
7    box compB ((B) -> (Q));
8    box compC ((C) -> (P));
9  }
10 connect compA || compB || compC;
11
12 net example {
13   net split
14   connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];
15
16   box examine ((P,Q) -> (A,B) | (Y,Z));
17
18   net compute {
19     net compAB (compAB_t)
20     connect compABC;
21
22     net syncPQ
23     connect [|{P},{Q}|] *{P,Q};
24   }
25   connect ( [{<T>} -> {}] .. compAB .. syncPQ ) !!<T>;
26 }
27 connect (tag .. split .. compute .. examine) *{Y,Z};
```

Figure 2.1: Running example to illustrate the transformation from S-Net to VR-Net

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 synchro-

nised 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.

2.3 Transformation of S-Net into VR-Net

2.3.1 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 specifications into the S-Net$_{\text{flat}}$ intermediate representation format, defined in Fig. 2.2.

$SNetExpr_{flat}$	$\Rightarrow$	*BoxName*
	$\mid$	*NetName*
	$\mid$	*Sync*
	$\mid$	*Filter*
	$\mid$	*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 2.2: Grammar of S-Net$_{\text{flat}}$

The significant difference between S-Net 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. 2.3 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.

```
1  //! snet flat
2
3  net tag ({A,B} -> {A,B,<T>});
4
5  net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {
6    box compA ((A) -> (P));
7    box compB ((B) -> (Q));
8    box compC ((C) -> (P));
9
10   net _SL
11   connect compA;
12
13   net _SR {
14     net _SL
15     connect compB;
16
17     net _SR
18     connect compC;
19   }
20   connect _SL || _SR;
21 }
22 connect _SL || _SR;
```

Figure 2.3: Running example after topology flattening
(continued on next page)

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. 2.3 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. 2.3 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. 2.4.

2.3.2 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 [1]. More formally, the type inference system turns S-Net$_{\text{flat}}$ code into VR-Net code, as defined in Fig. 2.5. In fact, the syntactic differences between S-Net$_{\text{flat}}$ and VR-Net 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 [1], 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.

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 VR-Net. Differences between annotated and inferred type information may be exploited for optimisation purposes at a subsequent compilation stage.

Fig. 2.6 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 [1].

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 {
  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 2.3: Running example after topology flattening
(continued from previous page)

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

Figure 2.4: Creating network names from location in topology expression

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

Figure 2.5: Grammar of VR-Net

```
1  //! snet typed
2
3  net tag ({A,B} -> {A,B,<T>});
4
5  net compABC ({A} -> {P}, {B} -> {Q}, {C} -> {P}) {
6    box compA ((A) -> (P));
7    box compB ((B) -> (Q));
8    box compC ((C) -> (P));
9
10   net _SL ({A} -> {P})
11   connect compA;
12
13   net _SR ({B} -> {Q}, {C} -> {P}) {
14     net _SL ({B} -> {Q})
15     connect compB;
16
17     net _SR ({C} -> {P})
18     connect compC;
19   }
20   connect _SL || _SR;
21 }
22 connect _SL || _SR;
```

Figure 2.6: Running example after type inference
(continued on next page)

```
1  net example ({A,B} -> {Y,Z}) {
2    net split ({A,B,<T>} -> {A,<T>} | {B,<T>})
3    connect [{A,B,<T>} -> {A,<T>}; {B,<T>}];
4
5    box examine ((P,Q) -> (A,B) | (Y,Z));
6
7    net compute ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
8      net compAB ({A} -> {P}, {B} -> {Q}, {C} -> {P})
9                 ({A} -> {P}, {B} -> {Q})
10     connect compABC;
11
12     net syncPQ ({P} -> {P,Q}, {Q} -> {P,Q}) {
13       net _ST ({P} -> {P} | {P,Q}, {Q} -> {Q} | {P,Q})
14       connect [| {P}, {Q}|];
15     }
16     connect _ST *{P,Q};
17
18     net _IS ({A,<T>} -> {P,Q}, {B,<T>} -> {P,Q}) {
19       net _SL ({<T>} -> {})
20       connect [{<T>} -> {}];
21
22       net _SR ({A} -> {P,Q}, {B} -> {P,Q})
23       connect compAB .. syncPQ;
24     }
25     connect _SL .. _SR;
26   }
27   connect _IS !!<T>;
28
29   net _ST ({A,B} -> {A,B} | {Y,Z}) {
30     net _SR ({A,B,<T>} -> {A,B} | {Y,Z}) {
31       net _SR ({A,<T>} -> {A,B} | {Y,Z}, {B,<T>} -> {A,B} | {Y,
             (cont.)Z}) {
32         net _SR ({P,Q} -> {A,B} | {Y,Z})
33         connect examine;
34       }
35       connect compute .. _SR;
36     }
37     connect split .. _SR;
38   }
39   connect tag .. _SR;
40 }
41 connect _ST *{Y,Z};
```

Figure 2.6: Running example after type inference
(continued from previous page)

Chapter 3

Conclusion

In this document we described the key requirements, the format, and the implementation of VR-Netas used in the **Advance** project. VR-Net encodes the essential information of a network description that allows for automatic aggregation and composition of statistical properties of applications written in S-Net. An automatic transformation process from S-Net to VR-Net has been developed and is detailed in this document. The key components of this transformation are the topology flatter and the type inference system. Both of these components have been described and their operation has been illustrated by means of a running example.

Bibliography

[1] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.