Center for Embedded Computer Systems
University of California, Irvine

System-On-Chip Network Modeling Style Guide

Dongwan Shin
Junyu Peng
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-23
July 31, 2004

System-On-Chip Network Modeling Style Guide

Dongwan Shin
Junyu Peng
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-23
July 31, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

http://www.cecs.uci.edu

Abstract

The SCE system-level design flow consists of a series of refinement steps that gradually transform an abstract specification model into an architecture model, a network model, a communication model and finally a detailed implementation model. The transformations can be automated with model refinement tools.

This report describes the styles of network models automatically generated by the network refinement tool. Therefore it can help designers understand the network models. It can also be used as a reference maunal if designers want to write their own network models that are valid input to the communication refinement tool.

Contents

List of Figures

System-On-Chip Network Modeling Style Guide

D. Shin, J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski.
Center for Embedded Computer Systems
University of California, Irvine

July 31, 2004

Abstract

The SCE system-level design flow consists of a series of refinement steps that gradually transform an abstract specification model into an architecture model, a network model, a communication model and finally a detailed implementation model. The transformations can be automated with model refinement tools.

This report describes the styles of network models automatically generated by the network refinement tool. Therefore it can help designers understand the network models. It can also be used as a reference maunal if designers want to write their own network models that are valid input to the communication refinement tool.

1 Introduction

System design in the SoC approach takes an initial specification of the system down to an actual implementation through a series of interactive and automated steps. Starting from a purely functional description of the desired system behavior, an implementation of the design on a heterogeneous system architecture with multiple processing elements (PEs) connected through system busses is produced at the end of the design flow.

This report describes and defines guidelines and rules for developing SpecC based system models in general, and as input to the SoC tools in particular.

1.1 SoC Design Flow

In the SoC design flow (Figure 1), five design models are used to represent the design at different abstraction levels. The design models are executable so that they can be simulated to verify the correctness of the design and obtain design performance metrics at each design step.

The most abstract model is the *specification model* that serves as the input to SoC tools. Specification model is a pure functional model that captures the functionality of the desired design. It should not contain any implementation details.

The *architecture model* is the output of architecture exploration. It reflects the overall computation architecture consisting of processing elements (PEs). The architecture model encapsulates the communication between PEs through abstract message-passing channels.

After network exploration a *network model* is produced to reflect the communication network chosen for the design. It represents the allocation and selection of network stations and the links between them. While the communication is end-to-end between PEs in the architecture model, it is refined into point-to-point in the network model.

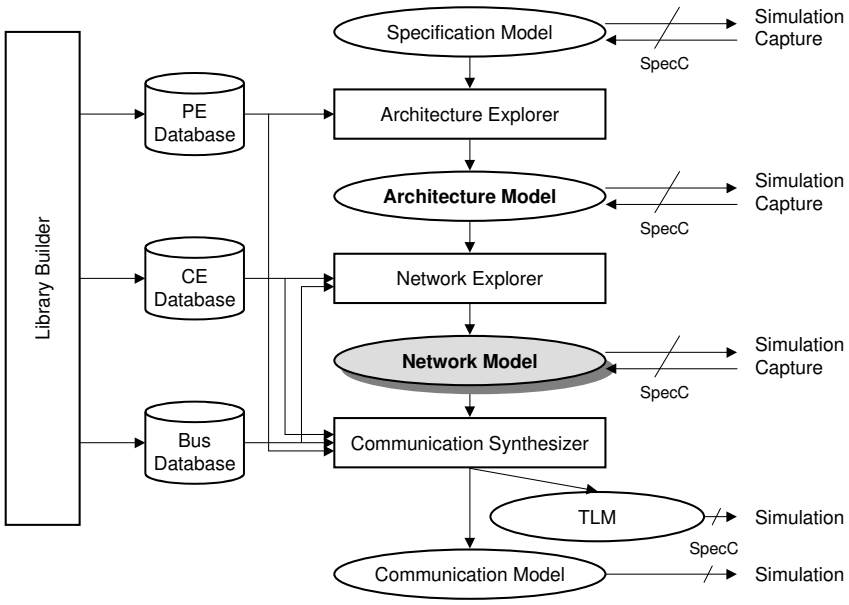

Figure 1: SoC design flow.

Finally, the *communication model* incorporates bus protocols into the model. The communication model can be pin-accurate or transaction-level. The *transaction level model* abstracts away the pin-accurate protocol details.

All the models are captured in SpecC language and they have to adhere to the syntax and semantics of the SpecC language. Designers only need to write the specification model for the design and then use the tools to automatically generate the lower level models. SoC tools also support partial specification, which allows designers to start with a specification with incomplete computational functionality. Later designers can modify the computation part of the automatically generated models. However, for the modified models to be valid for the SoC tools, they must follow certain modeling rules. This report in particular defines the modeling style required for the SoC network model, which is highlighted in the figure. [4], [5], [7] and [6] focus on the modeling styles of the other models.

1.2 SpecC Language

The SoC design flow is supported by the SpecC system-level design language ([1]). The SpecC language as an example of a modern system-level design language (SLDL) was developed under support and control of the SpecC Technology Open Consortium (STOC) ([2]) to satisfy all the requirements for an efficient formal description of the models in the SoC design flow. It supports behavioral and structural views and contains features for describing a design at all levels of abstraction.

In the SoC design flow, all five models of the design process starting with the specification model and down to the implementation model are described in the SpecC language. One common language removes the need for tedious translation. Furthermore, all the models in SpecC are executable which allows for validation through simulation while reusing one single testbench throughout the whole design flow. In addition, the formal nature of the models enables application of formal methods, e.g. for verification or equivalence checking.

Note that this report is not intended to be a tutorial of SpecC language and we assume that the reader of this report is familiar with the language. This report can be used for two purposes. First, it can help users understand the meaning of the automatically generated network models by the network explorer. Second, it can help users modify the automatically generated network models such that they can be accepted by the communication synthesizer ([**?**]).

The rest of the report is organized as follows. Section 2 presents the overall structure of a network model. The major elements of a network model are described one by one in detail. Section 3 describes the communication channels allowed in the network model. Section 4 describes the modeling of shared memories in the network model. Section 5 describes the guidelines to model processing elements. Section 6 describes the modeling of communication elements, such as bridges and transducers in the network model. Finally, Section 7 describes the rules on how to compose the element together to form an valid network model.

2 An Overview of Network Model

Figure 2 and Figure 3 show an example template for a valid network model. A network model has to be an executable SpecC model, i.e. it has to define a `Main` behavior. A network model consists of a testbench that surrounds the actual design to be implemented. A testbench consists of stimulating (`Stimulus`) and monitoring (`Monitor`) behaviors that are executing concurrently with the design (`Design`) in the top-most `Main` behavior, and that drive the design under test and check the generated output against known golden outputs.

The actual design to be implemented is modeled by the *design behaviors*, such as behavior `Design` and those composed hierarchically inside `Design` in Figure 3. Design behaviors form a hierarchy tree by their composition relations. The root of the tree, for example `Design` in Figure 3, is called the *top-level design behavior*.

Note that the modeling rules and restrictions defined in this report apply only to the design behaviors since the testbench behaviors will not be considered and touched by SoC tools. Therefor the testbench can be freely described using any valid SpecC code. For example, while the code of the design to be implemented has to be available completely in SpecC source form, the testbench can link against external translation units (libraries) for additional functionality.

In general it is hard for SoC tools themselves to find out which behaviors are testbench behaviors and which are actual design behaviors. This distinction is made by the designers attaching a predefined annotation to the network model.

Rule 1 *A network model has an annotation `_SER_TOPLEVEL`, which contains the name of the top-level design behavior of the model.*

For the example shown in Figure 3, the annotation would look like the following:

```
note _SER_TOPLEVEL = ''Design'';
```

Once the top-level design behavior is specified, the SoC tools are able to figure out all other design behaviors.

If we zoom inside the top-level design behavior of a network model, we can identify a set of finer model elements which are used to capture both the computation architecture and the communication network.

- *PE behaviors* are used to model the processing elements allocated to perform the desired computation;
- *Memory behaviors* are used to model the memories allocated to store data shared by PEs;
- *Bridge* and *transducer behaviors* are used to model the communication elements needed to interface between different communication protocols.

```
import "c_double_handshake";

behavior Stimulus(i_sender input) {                    // Stimuli creator
  void main(void) {
    // while(...) { ... ; input.send(...) ; ... }
  }
};

behavior Monitor(i_receiver output) {                  // Output monitor
  void main(void) {
    // while(...) { ... ; output.receive(...) ; ... }
  }
};

behavior Design(i_receiver input, i_sender output) {   // System design
  // ...

  void main(void) {
    // fsm { ... }
  }
};

behavior Main() {                                      // Top level
  c_double_handshake input, output;

  Stimulus stimulus(input);
  Design   design(input, output);
  Monitor  monitor(output);

  int main(void) {
    par {
      stimulus.main();
      design.main();
      monitor.main();
    }
  }
};
```

Figure 2: Network model top-level code.

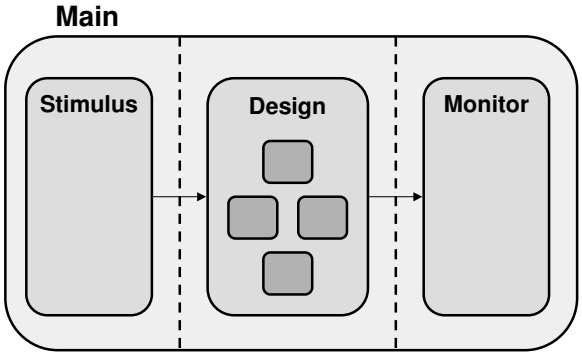

Figure 3: Network model top-level structure.

- *Link channels* are used to model the connection between the processing elements and the communication elements.

Rule 2 *The top-level design behavior of a network model has three annotations attached to it:* `_AR_PES`*,* `_NR_CES` *and* `_CR_BUSSES`*, which contain the names, types and other attributes of processing elements, communication elements and busses allocated for the design.*

In the following sections, we will define these model elements one by one before we describe how to put them together to form a valid network model.

3 Link Channels

In the network models, SpecC channels are used to represent the links that connect processing elements (see Section 5) and communication elements (see Section 6). A link only allows sequential and uni-directional transactions. Transactions that may occur at the same time must go through different links. The links in the network models are logical since different links may be later implemented on the same physical bus.

Network links can be classified based on the information they convey: *data links* are used for data transfer while *synchronization links* are used for pure synchronization with no data involved. To enable automatic communication synthesis, both the data links and the synchronization links have to be represented with certain types of channels defined in the library.

Rule 3 *A data link is represented with an instance of untyped* `c_double_handshake` *channel as defined in SpecC Language Reference Manual.*

```
interface i_sender {
  void send(const void *d, unsigned long l);
};

interface i_receiver {
  void receive(void *d, unsigned long l);
};

interface i_tranceiver {
  void send(const void *d, unsigned long l);
  void receive(void *d, unsigned long l);
};
```

Figure 4: Interfaces implemented by `c_double_handshake` channel.

The `c_double_handshake` channel encapsulates un-buffered type-less data transfer. It implements three interfaces: `i_sender`, `i_receiver` and `i_tranceiver` as shown in Figure 4.

Rule 4 *A synchronization link is represented with an instance of* `c_handshake` *channel as defined in SpecC Language Reference Manual.*

The `c_handshake` channel encapsulates one-way handshake synchronization that does not need to carry real data. It implements two interfaces: `i_send` and `i_receive` as shown in Figure 5.

For more information about rules and examples on using `c_double_handshake` and `c_handshake` channels, the reader is referred to SpecC Language Reference Manual ([1]). Note that additional rules will be presented in Section 7 for connecting the link channel instances to PE ports.

```
interface i_send {
  void send(void);
};

interface i_receive {
  void receive(void);
};
```

Figure 5: Interfaces implemented by `c_handshake` channel.

4 Memories

In the network models, SpecC behaviors are used to model memory components which store data. For PE behaviors to access the stored data, a memory behavior has to implement a *memory interface* which provides memory read and write methods. Since the data being transferred in the network models is untyped, the memory interface should not define any type-specific methods. An example code of memory behavior is shown in Figure 6.

Rule 5 *A memory behavior implements exactly one interface, which has two methods with exactly the following signatures:*

```
void read(unsigned long offset, void *d, unsigned long size);
void write(unsigned long offset, void *d, unsigned long size);
```

The `read` method reads *size* bytes from the memory location starting at *offset* and puts them in the variable which is pointed by *d*. Conversely, the `write` method writes *size* bytes data pointed by *d* to the memory location starting at *offset*.

Rule 6 *A memory behavior must not have any ports, channel instances and sub-behavior instances. It has only one variable `mem`, which is a C `struct` that exactly describes the actual sizes (in terms of number of bytes) and the locations of all variables stored in the memory.*

Note that the actual sizes for a same data type may vary from processor to processor. For example, a `short` may occupy two bytes in one processor while four bytes in another. Assigning locations to variables has to take the alignment factor into account: can a variable start at byte, word or double-word boundary? If necessary, padding has to be used to make sure that variables align correctly. In the example shown in Figure 6, the two variables stored in the memory (`x` and `y`) are packed into the struct variable `mem`. Here we assume the size of `short` type is two bytes and the size of `float` type is four bytes. We also assume that a variable can start at word boundary.

Note that the body of the `main()` method is empty, i.e. it does not do anything. Other than the `main()` method and the memory interface methods, the memory behavior should not implement any other methods.

Rule 7 *Each memory behavior must have a `_PE_BF_MODEL` annotation, which contains the behavior name of the memory's bus-functional model stored in the database.*

In order to help the communication synthesizer locate the bus-functional model of the memory, an annotation `_BF_MODEL` must be attached to the memory behavior. For example,

```
Mem1._PE_BF_MODEL = ``Samsung\_BF'';
```

tells the communication synthesizer that the behavior name of the memory's bus-functional model is *Samsung_BF*.

```
#include <string.h>

interface i_Mem1
{
  void read(unsigned long, void*, unsigned long);
  void write(unsigned long, const void*, unsigned long);
};

behavior Mem1 implements i_Mem1
{
  struct {
    char x[2];      // short x;
    char y[40];     // float y[10];
  } mem;

  void write(unsigned long ofs, const void* data, unsigned long len)
  {
    char *ptr;
    ptr = (char*) &mem;
    memcpy(ptr + ofs, data, len);
  }

  void read(unsigned long ofs, void *data, unsigned long len)
  {
    char *ptr;
    ptr = (char*) &mem;
    memcpy(data, ptr + ofs, len);
  }

  void main(void)
  {
  }
};
```

Figure 6: An example of memory behavior.

5 Processing Elements

In a network model each processing element is represented by a *PE behavior*. The PE behavior specifies the functionality to be implemented by the PE rather than the internal structure of the PE. The basic functionality of a PE is data processing or computation.

In general, a PE performs computation on typed data, such as `integer` and `float`. However, as we have defined earlier, PEs in the network models must exchange data through `c_double_handshake` channels which only provide type-less data transfer. Therefore PE behaviors in the network models must model the formatting between typed data and type-less data.

In a network model, transducers may be needed to interface two busses with incompatible protocols. It may happen that data transfers between two PEs have to go through a transducer when the communicating PEs are connected to two busses with incompatible protocols. Basically, the transducer receives the data from one PE, buffers it and then send it out to the other PE. Usually a transducer has a finite amount of internal buffer, which limits the size of data exchanged between the transducer and the PEs in one transaction. In this case, PEs have to break data into smaller *packets* to avoid overflow in the transducer buffer. Therefore PE behaviors in the network models must also model the packetization process, if necessary.

5.1 Two-Shell Structure

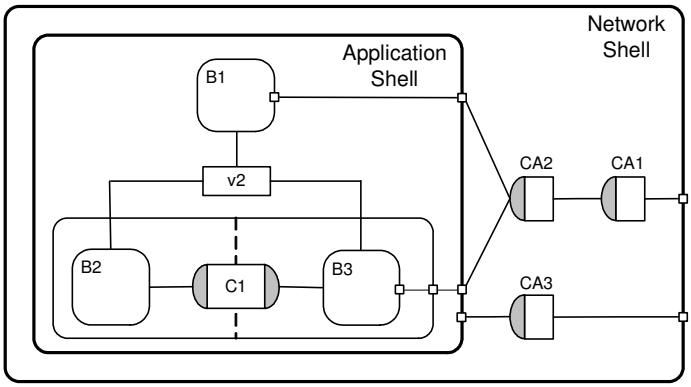

Figure 7: Two-shell PE structure.

In order to separate the modeling of application domain from that of network domain support, a two-shell modeling style (Figure 7) has to be followed in a PE behavior.

The inner application shell behavior encapsulates the computation required by the application executed on the PE. In general, the application shell behavior is hierarchically composed of smaller behaviors (`B1`, `B2` and `B3`), where each contains a piece of the compuation assigned to the PE. To model inter-behavior communication inside the application shell, both channels (`C1`) and variables (`v2`) can be connected to the behavior ports (small squares at behavior boundary). The modeling styles inside the application shell can be found in SpecC specification model reference manual.

However, the application shell behavior can only have interface type ports and no variable ports. Furthermore, only certain interface types are allowed for the ports.

Rule 8 *An application shell behavior has only interface ports and no variable ports. The interface types allowed are as follows:*

(a) `i_sender` *(typed or un-typed)*

(b) `i_receiver` *(typed or un-typed)*

(c) `i_tranceiver` *(typed or un-typed)*

(d) `i_send`

(e) `i_receive`

(f) memory interface (Section 4)

```
import "c_double_handshake";

#include <c_typed_double_handshake.sh>

/* create "float" type interface "i_data1_tranceiver" */
DEFINE_I_TYPED_TRANCEIVER(data1, float)

/* create "float" type interface "i_data1_sender" */
DEFINE_I_TYPED_SENDER(data1, float)

/* create "float" type interface "i_data1_receiver" */
DEFINE_I_TYPED_RECEIVER(data1, float)


channel adapter (i_tranceiver link) implements
        i_data1_sender, i_data1_receiver
{
  void receive(float *data1)
  {
    link.receive(data1, sizeof (float));
  }

  void send(float data1)
  {
    link.send(&data1, sizeof(float));
  }
};
```

Figure 8: Example code of an adapter channel.

The example shown in Figure 8 illustrates how to create typed interfaces that send and/or receive a `float` type data (line 5 to 12).

The network shell is needed to format typed data in the application into stream of untyped data transferred over the network. It may also be needed to break the byte stream into packets that will be routed over the network if the data transfer goes through a transducer. The network shell is modeled as a hierarchical behavior that has an instantiation of the application shell behavior. It may also has a set of adapter channel instances that perform the and typed/untyped conversion and/or packetization. An example of an adapter channel that converts `float` data transfer into untyped data transfer is shown in Figure 8 (line 15 to 27).

Rule 9 *The network shell behavior contains exactly one behavior instance, whose type is an application shell behavior. It may contain a set of network shell adapter channel instances.*

Similar to the application shell behavior, the network shell behavior has only interface ports and only untyped interface types are allowed. These interface ports will be connected to the link channel instances defined in Section 3.

Rule 10 *A network shell behavior has only interface ports and no variable ports. The interface types allowed are as follows:*

(a) `i_sender` *(un-typed)*

(b) `i_receiver` *(un-typed)*

(c) `i_tranceiver` *(un-typed)*

(d) `i_send`

(e) `i_receive`

(f) memory interface (see Section 4)

Rule 11 *The network shell behavior of a software PE has annotation* `_PE_BF_MODEL` *attached to it, which contains the behavior name of the PE's bus-functional model stored in the database.*

5.2 Memory-Mapped IO

A design may use both software and hardware PEs. Software PEs are general purpose programmable processors while hardware PEs are application specific hardware units that need to be synthesized. One common practice for communication between a processor and a hardware device is by means of memory mapped-IO. Basically, the hardware unit uses it internal memory, for example, registers as IO ports that are mapped to the processor's memory space. As such, these registers can be accessed by the processor as if they were memories.

For memory-mapped IO, the internal memory of a hardware PE can be modeled as a memory behavior described in Section 4. (Note that memory-mapped IO modeling does not apply to the software processors.) In this case, a hardware PE in the network model is described as a combination of a PE behavior defined in Section 5.1 and a memory behavior defined in Section 4. More specifically, the memory behavior instantiated inside the application shell implements an untyped memory interface, which is propagated to the application shell behavior and the network shell behavior.

However, the application code executed by the hardware PE itself may also need to access the registers as typed data. Obviously, the local accesses do not go through the network. Therefore in this case, the memory behavior has to implement an additional typed memory interface for the local access. An example code of a hardware PE behavior with internal memory is shown in Figure 9.

6 Communication Elements

Two kinds of communication elements, namely bridges and transducers, can be modeled in the network model.

```
interface i_untyped_Mem1 {
  void read(unsigned long, void*, unsigned long);
  void write(unsigned long, const void*, unsigned long);
};

interface i_typed_Mem1 {
  short read_x(void);
  void write_x(short);
};

behavior Mem1 implements i_untyped_Mem1, i_typed_Mem1
{
  // code ommited for space consideration
};

behavior Beh(i_typed_Mem1);

behavior AppShell implements i_untyped_Mem1
{
  Mem1 M1;
  Beh  B1(M1);

  void main(void)  {  B1.main();  }

  void read(unsigned long ofs, void* d, unsigned long len)
  {     M1.read(ofs, d, len);  }

  void write(unsigned long ofs, const void* d, unsigned long len);
  {     M1.write(ofs, d, len); }
};

behavior NetShell implements i_untyped_Mem1
{
  AppShell  App;

  void main(void)  {  App.main();  }

  void read(unsigned long ofs, void* d, unsigned long len)
  {     App.read(ofs, d, len); }

  void write(unsigned long ofs, const void* d, unsigned long len);
  {     App.write(ofs, d, len); }
};
```

Figure 9: Example code of memory-mapped IO modeling.

```
import "i_sender";
import "i_mem1";

interface i_sender_L1
{
  void send_L1(void*, unsigned long);
};

interface i_sender_L3
{
  void send_L3(void*, unsigned long);
};

behavior Bridge(i_sender L1, i_mem1 L2, i_sender L3)
        implements i_sender_L1, i_mem1, i_sender_L3
{
  void main(void)
  {
  }

  void send_L1(void *d, unsigned long len)
  {
    L1.send(d, len);
  }

  void read(unsigned long ofs, void *d, unsigned long len)
  {
    L2.read(ofs, d, len);
  }

  void write(unsigned long ofs, const void *d, unsigned long len)
  {
    L2.write(ofs, d, len);
  }

  void send_L3(void *d, unsigned long len)
  {
    L3.send(d, len);
  }
};
```

Figure 10: An example of `bridge` behavior.

6.1 Bridges

In the network model, a bridge is modeled as a SpecC behavior. An example of bridge behavior is shown in Figure 10.

Rule 12 *A bridge has a set of ports of following types:*

(a) `i_sender` *(un-typed)*

(b) `i_receiver` *(un-typed)*

(c) `i_tranceiver` *(un-typed)*

(d) `i_send`

(e) `i_receive`

(f) memory interface (see Section 4)

Due to the defined port types, A bridge can be connected to either `c_double_handshake`, `c_handshake` or memory instances.

Rule 13 *A bridge usually implements a set of interfaces. The methods of the implemented interfaces must have the same argument list as the one of the port interface. In addition, the name of the interface methods must starts with either "send" or "read". Therefore, the methods must have one of the following signatures:*

```
void  send_XXXX(void *, unsigned long);
void  receive_XXXX(void *, unsigned long);
void  send_XXXX(void);
void  receive_XXXX(void);
void  read_XXXX(unsigned long, void *, unsigned long);
void  write_XXXX(unsigned long, void *, unsigned long);
```

Note that XXXX represents an user-defined string.

The bridge shown in Figure 10 has three ports that can be connected to two `c_double_handshake` channel instances and a memory behavior respectively. From the modeling point of view, a bridge is just an transparent adapter that calls the same kind of methods without modifying the data transfer.

Rule 14 *A bridge behavior has annotation* `_CE_BF_MODEL` *attached to it, which contains the behavior name of the bridge's bus-functional model stored in the database.*

When a bridge has only a port of memory interface type and it implements the same interface by itself, this bridge is actually a meory controller.

The bridge behaviors are later replaced with a bus-functional model during communication synthesis. Therefore, it must have an annotation to inform the communication synthesizer the name of its corresponding bus-functional model.

6.2 Transducers

During network exploration, multiple busses may be allocated to realize the communication between PEs. In the network model, transducer behaviors may be needed to represent the communication elements that interface two different communication protocols through buffecan moring.

6.2.1 Simple case: Asynchronous Transducers

A transducer may need to hand data transfer or data-less synchronization that involves different protocols on its two sides. We will use `buffer` behavior to model the handling of data transfer by the transducer and `repeater` behavior to model the handling of synchronization by the transducer. A transducer behavior can then be modeled as a `par` composition of a set of `buffer` and `repeater` behavior instances.

```
#define PACKET_SIZE 1

import "i_sender";
import "i_receiver";

behavior Buffer(i_receiver L, i_tranceiver R)
{
  char buffer[PACKET_SIZE];

  void main(void)
  {
    while(1)
    {
      L.receive(&buffer[0], PACKET_SIZE);
      R.send(&buffer[0], PACKET_SIZE);
    }
  }
};
```

Figure 11: An example of `buffer` behavior.

As shown in Figure 11, a `buffer` behavior has two ports of interface type, `i_sender` and `i_receiver`. (We will see later that these two ports are to be mapped to `c_double_handshake` channel instances.) It also has a buffer that is the size of a packet. The `main()` method is an endless `while` loop, where data is received from the `i_receiver` port and stored in the buffer before it is sent out through the `i_sender` port.

```
import "i_send";
import "i_receive";

behavior Repeater(i_receive L, i_send R)
{
  void main(void)
  {
    while(1)
    {
      L.receive();
      R.send();
    }
  }
};
```

Figure 12: An example of `repeater` behavior.

Rule 15 *A `buffer` behavior is a leaf behavior that has two ports, one being `i_sender` type and the other being `i_receiver` type.*

As shown in Figure 12, a `repeater` behavior has two ports which are `i_send` and `i_receive` interface type respectively. (We will see later that these two ports are to be mapped to `c_handshake` channel instances.) The `main()` method is an endless `while` loop, where synchronization is received from the `i_receive` port before it is passed through the `i_send` port.

Rule 16 *A `repeater` behavior is a leaf behavior that has two ports, one being `i_sender` type and the other being `i_receiver` type.*

A transducer behavior is a hierarchical behavior that may consist of one or more `buffer` and `repeater` behavior instances executing in parallel.

```
behavior Transducer(i_receiver L1, i_sender R1,
                    i_sender L2, i_receiver R2,
                    i_send L3, i_receive R3)
{
  Buffer b1(L1, R1);
  Buffer b2(R2, L2);
  Repeater r1(R3, L3);

  void main(void)
  {
    par
    {
      b1.main();
      b2.main();
      r1.main();
    }
  }
};
```

Figure 13: An example code of `transducer` behavior.

Rule 17 *A transducer behavior is a hierarchical behavior that has a set of ports and a set of `buffer` and `repeater` behavior instances. The port type can be either `i_sender`, `i_receiver`, `i_sender`, or `i_receiver`.*

Rule 18 *Inside the transducer behavior, no ports of the transducer behavior is port-mapped to more than one sub-behavior.*

6.2.2 More Complex Case: Synchrounous Transducers

The transducers presented in Section 6.2.1 are called *asynchronous* because they do not automatically preserve the synchronicity required by the two communicating parties. For instance, a double-handshake data transfer between two parties requires both of them to meet at the same time to start the transfer. However, when an asynchronous transducer is inserted, the double-handshake data transfer is splitted into two sequential transfers, which lack the synchronicity: the sending party deposits a data item to the transducer then proceeds without worrying about if the receiving party is there to retrieve the data. In order to retain the

```
/* uni-directional synchronous data transfer */
behavior TX(i_tranceiver L1A, i_tranceiver L1B)
{
  void main(void)
  {
    long int size1, size2;
    unsigned char data[PACKET_SIZE];

    while(1) {
      L1A.receive(&size1, sizeof(long int));
      L1B.receive(&size2, sizeof(long int));

      L1A.receive(&data[0], size1);
      L1B.send(&data[0], size2);
    }
  }
};

/* bi-directional synchronous data transfer */
behavior TX(i_tranceiver L1, i_tranceiver L2)
{
  void main(void)
  {
    short int dir1, dir2;
    long int size1, size2;
    unsigned char data[PACKET_SIZE];

    while(1) {
      L1.receive(&dir1, sizeof(short int));
      L1.receive(&size1, sizeof(long int));
      L2.receive(&dir2, sizeof(short int));
      L2.receive(&size2, sizeof(long int));

      if(dir1 == RECEIVE && dir2 == SEND)
      {
        L2.receive(&data[0], size2);
        L1.send(&data[0], size1);
      }
      else
      {
        L1.receive(&data[0], size1);
        L2.send(&data[0], size2);
      }
    }
  }
};
```

Figure 14: An example of synchronous transducer for synchronous communication.

desired synchronicity, additional meta-data, such as a flag, has to be sent back and forth between the two parties.

In this section, we present another approach that uses *synchronous* transducers, which automatically preserve the synchronicity required by the data transfers being implemented. As a result, the synchronous transducers tend to be more costly than the asynchronous transducers which merely perform transparent data relay. However, the synchronous transducers avoid transfering meta-data for end-to-end synchronization thus potentially reduce communication traffic.

In this report, we present two types of synchronous transducers, one for implementing synchronous data transfers (double-handshake) and the other for implementing asynchronous (queue) data transfers.

Synchronous Communication Synchronous communication requires the sending and receiving parties meet at the same time for the data transfer. Synchronous communication between two end points can be modeled with a SpecC double-handshake channel that connects them. The double-handshake channel allows both uni-directional and bi-directional transfers. The example code of synchronous transducers for both uni- and bi-directional synchronous communication is shown in Figure 14.

Asynchronous Communication Asynchronous communication uses additional storage to buffer transferred data therefore it does not need the sending and receiving parties to meet at the same time. (Note that as an extreme case, asynchronous communication degenerates to synchronous communication if buffer size is zero.) Asynchronous communication between two end points can be modeled with a SpecC queue channel that connects them. Same as the double-handshake channel, a queue channel also allows uni- and bi-diretional transfers. The example code of synchronous transducers for both uni- and bi- directional asynchronous communication is shown in Figure 15 and Figure 16, respectively.

```
behavior TX1(i_tranceiver L1, i_tranceiver L2)
{
  void main(void)
  {
    long int size1, size2;
    unsigned char data[PACKET_SIZE];

    while(1) {
      L1.receive(&size1, sizeof(long int));
      L1.receive(&data[0], size1);

      L2.receive(&size2, sizeof(long int));
      L2.send(&data[0], size2);
    }
  }
};
```

Figure 15: An example of synchronous transducer for asynchronous communication.

Note that transducers can be synthesized in the same way as the hardware PEs during the back-end process.

```
behavior Left(i_tranceiver L2A, i_sender cq1, i_receiver cq2)
{
  void main(void)
  {
    short int sr;
    long int size;
    long int data1;
    short int data2;

    while(1) {
      L2A.receive(&sr, sizeof(short int));
      L2A.receive(&size, sizeof(long int));
      if(sr == SEND)
      {
        L2A.receive(&data1, size);
        cq1.send(&data1, size);
      }
      else
      {
        cq2.receive(&data2, size);
        L2A.send(&data2, size);
      }
    }
  }
};

behavior Right(i_tranceiver L2B, i_receiver cq1, i_sender cq2)
{
  void main(void)
  {
    short int sr;
    long int size;
    long int data1;
    short int data2;

    while(1) {
      L2B.receive(&sr, sizeof(short int));
      L2B.receive(&size, sizeof(long int));
      if(sr == SEND)
      {
        L2B.receive(&data2, size);
        cq2.send(&data2, size);
      }
      else
      {
        cq1.receive(&data1, size);
        L2B.send(&data1, size);
      }
    }
  }
};

behavior TX2(i_tranceiver L2A, i_tranceiver L2B)
{
  c_queue cq1(500ul), cq2(500ul);

  Left left(L2A, cq1, cq2);
  Right right(L2B, cq1, cq2);

  void main(void)
  {
    par {
          left.main();
          right.main();
        }
  }
};
```


Figure 16: An example of synchronous transducer for asynchronous communication.

7 Top-Level Design Behavior

In previous sections, we have defined the set of model elements in a network model. In this section we present the following rules on connecting the elements together to form a network model that is a valid input to the communication synthesizer. An example network model is shown in Figure 17.

```
import "c_double_handshake";

interface i_Memory {
  void read(unsigned long, void*, unsigned long);
  void write(unsigned long, void*, unsigned long);
};

behavior Memory implements i_Memory;
behavior MemCntl(i_Memory) implements i_Memory;
behavior Transducer(i_receiver, i_sender);
behavior CPU(i_sender, i_Memory);
behavior HW(i_receiver);

behavior Design(void)
{
  c_double_handshake L1;
  c_double_handshake L2;

  Memory      M;
  MemCntl     B(M);
  Transducer  T(L1, L2);
  CPU         C(L1, B);
  HW          H(L2);

  void main(void)
  {
    par {
      M.main();
      B.main();
      T.main();
      C.main();
      H.main();
    }
  }
};
```

Figure 17: Example of a top-level design behavior code.

Rule 19 *The top-level design behavior of the network model is a hierarchical behavior that is composed of following:*

(a) instances of link channels as defined in Section 3,

(b) instances of PE behaviors as defined in Section 5,

(c) instances of memory behaviors as defined in Section 4, and

(d) instance of CE behaviors as defined in Section 6.

Rule 20 *The top-level behavior has exactly one method, the* `main()` *method, which contains exactly one statement that is of* `par` *type.*

By definition of a hierarchical behavior ([4]), each sub-behavior instance inside the top-level behavior can be called at most once in the `par` statement. For example, having two $PE1.main()$ calls in the `par` statement is not allowed.

Rule 21 *Inside the top-level design behavior, a link channel instance can be connected to only two PEs/CEs. Each PE or CE has at most one port mapped to a given link channel.*

Rule 22 *In the network model, at most one CE, either a bridge or a transducer is allowed between two PEs.*

Since a bridge implements interfaces, it can be mapped to PE ports like the link channels. A bridge also has a set of ports therefore it can also connect to a PE using link channels. When a bridge is connected to two PEs that communicate with each, the following rule has to be obeyed.

Rule 23 *Between two communicating PEs, one must be the master and the other be the slave. If there is a bridge between them, the bridge must be port-mapped to the master PE and be connected to the slave PE through link channel instances.*

Note that the top-level behavior may have a set of ports to communicate with the testbench behaviors. It is strongly recommended that the top-level behavior keep the same set of ports if the architecture model is derived from a specification model. The same interface of the top-level behavior enables re-use of testbench behaviors for simulation of the architecture model without any modification.

References

[1] R. Dömer, A. Gerstlauer and D. D. Gajski. *SpecC Language Reference Manual, Version 2.0*, SpecC Technology Open Consortium (STOC), Japan, December 2002.

[2] SpecC Technology Open Consortium. http://www.specc.org.

[3] SpecC Compiler V2.2.0, Center for Embedded Computer Systems, University of California, Irvine, June 2004.

[4] A. Gerstlauer, K. Ramineni, R. Dömer, D. Gajski. *System-on-Chip Specification Style Guide*, Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[5] J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Architecture Modeling Style Guide*, Technical Report CECS-TR-04-22, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[6] D. Shin, L. Cai, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Transaction-Level Modeling Style Guide*, Technical Report CECS-TR-04-24, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[7] D. Shin, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Communication Modeling Style Guide*, Technical Report CECS-TR-04-25, Center for Embedded Computer Systems, University of California, Irvine, July 2004.