Chapter 1

REFINING SYNCHRONOUS COMMUNICATION ONTO NETWORK-ON-CHIP BEST-EFFORT SERVICES

Zhonghai Lu, Ingo Sander, and Axel Jantsch
Department of Electronic, Computer and Software Systems
Royal Institute of Technology, Sweden
{zhonghai,ingo,axel}@imit.kth.se

Abstract We present a novel approach to refine a system model specified with perfectly synchronous communication onto a Network-on-Chip (NoC) best-effort communication service. It is a top-down procedure with three steps, namely, *channel refinement, process refinement,* and *communication mapping.* In channel refinement, synchronous channels are replaced with stochastic channels abstracting the best-effort service. In process refinement, processes are refined in terms of interfaces and synchronization properties. Particularly, we use *synchronizers* to maintain local synchronization of processes and thus achieve *synchronization consistency,* which is a key requirement while mapping a synchronous model onto an asynchronous architecture. Within communication mapping, the refined processes and channels are mapped to a NoC architecture. Adopting the *Nostrum* NoC platform as target architecture, we use a digital equalizer as a tutorial example to illustrate the feasibility of our concepts.

Keywords: Synchronous Model, Communication Refinement, Network-on-Chip

1. Introduction

For system design, a synchronous design style is attractive since it allows us to separate timing from function. The designer can focus on the design of the system functionality without being distracted by unnecessary low-level communication details. This also facilitates the verification task, which is a key activity at the system level. Later, *refinement* explores the implementation space under constraints, making design decisions and filling in implementation details. Network-on-Chip (NoC) is an emerging SoC paradigm aimed to cope with the scalability problem of various buses in order to connect tens or perhaps even hundreds of microprocessor-sized heterogeneous resources, such as processor cores, DSPs, FPGAs/ASICs, and memories. The complex integration is desired by ever-increasing functionality and enabled by the steady technology scaling. Nostrum [11–13] is our NoC architecture offering a packet-switched communication platform. To satisfy different performance/cost requirements, Nostrum provides two classes of communication services, namely, Best Effort (BE) and Guaranteed Bandwidth (GB) services. The BE service is connectionless where packets are routed without resource reservation. The GB service is connection-oriented where packets are delivered after enough bandwidth is reserved. It achieves better performance at the expense of higher cost.

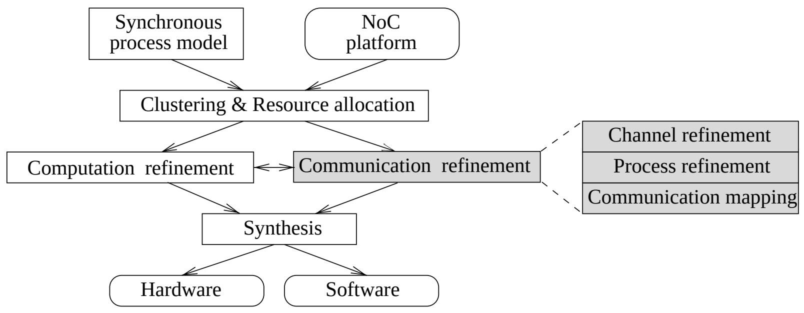


Figure 1.1. A NoC Design Flow

In this work, we are interested in mapping a system specified as a synchronous model onto a NoC. To this end, we propose a NoC design flow shown in Fig. 1.1 where we concentrate on the communication problem. There are three communication-related tasks: *clustering & resource allocation*, *communication refinement*, and *synthesis*. The clustering flattens the hierarchy in the model and groups processes into new processes with perhaps coarser granularity. With resource allocation, the grouped processes are allocated to network nodes, either HW or SW execution resources. Communication refinement bridges the gap between the communication model in the specification and the NoC communication implementation via adapters. With synthesis, these processes and adapters are synthesized into HW and/or SW.

We address the *communication refinement* that starts from a synchronous communication model and ends with the Nostrum NoC best-effort communication service. With the specification model, communication is perfectly synchronous with a global logical clock and cleanly separated from computation. With the NoC communication service, communication introduces variable delays and crosses multiple clock domains connected by a packet-switched network. Clearly the communication in the implementation domain is not synchronous, thus not consistent with that in the specification domain. Our contributions are (1) a novel approach to realize this communication refinement; (2) a classification of process synchronization properties as *strict*, *nonstrict*, *strong*, and *weak* synchronization in order to formally analyze processes' local synchronization requirement(s) (Section 5.2); (3) using *synchronizers* (synchronization adapters) to maintain synchronization consistency during refinement (Section 5.3). We will focus on the synchronization issue while keeping the process computation untouched. Note that, this synchronization issue lies at the system modeling level, not at the lower implementation levels such as shared memory synchronization using locks or semaphores, as well as message passing synchronization using blocking or nonblocking semantics. We assume that, after a clustering, the resulting processes, more precisely, process networks, are top-level entities. Each process may comprise a hierarchy of sub-processes which are intended to reside in a synchronous implementation domain. Besides, we consider that a resource maintains a local synchronous region. Consequently a process is to be mapped to one resource and one resource hosts exactly one process.

2. Related Work

Based on the isolation of communication from computation, a large body of work on communication refinement exists. Through the Virtual Component Interfaces (VCI) of the VSI Alliance [9], the COSY-VCC design flow [3] supports communication refinement from specification to performance estimation and to implementation. IPSIM [5] developed on top of SystemC 3.0 supports an object-oriented methodology and establishes two inter-module communication layers. The message box layer concerns generic and system-specific communication, while the driver layer implements higher level application-dependent communications. The SpecC methodology defines four levels of abstraction, namely at the specification, architecture, communication and implementation level, and the refinement transformations between them [6]. These works do not assume a synchronous specification.

With synchronous communication, latency insensitive theory [4] targets synchronized HW design where synchronization can still be achieved using relay stations even if interconnecting synchronous IP blocks experiences indefinite wire latencies; Desynchronization for SW design was addressed in [1]. Furthermore, some mathematical frameworks were developed to support refinement-based design methods. Benveniste et al. present a theoretical framework for modeling heterogeneous systems, and derive sufficient conditions to maintain semantic-preserving transformations when deploying a synchronous specification onto GALS and the loosely time-triggered architectures [2]. Another framework is proposed in [7] concerning the refinement of a polysynchronous specification, which allows the existence of multiple clocks instead of a single clock. All these works are complementary to ours but none of them provides a detailed refinement approach targeting a NoC platform.

3. Refinement Overview

3.1 The perfectly synchronous model

The synchronous modeling paradigm is based on an elegant and simple mathematical model, which is the ground of synchronous languages such as Esterel, Signal, Argos and Lustre. The basis is the perfect synchrony hypothesis, i.e., both computation and communication take no observable time. A system is modeled as a set of concurrent communicating processes via signals. Processes use ideal data types and assume infinite buffers. Signals are ordered sequences of events. Each event has a time slot as a slot to convey data. If the data contains useful information, the event is *present* and called a *token*; otherwise, the event is *absent* and modeled as a $\sqcup$ representing a clock tick. Each signal can be related to the time slots of another signal in an unambiguous way. The output events of a process occur in the same time slot as the corresponding input events. Moreover, they are instantaneously distributed in the entire system and are available to all other processes in the same slot. Receiving processes in turn consume the events and emit output events again in the same time slot. The medium a signal passes can thus be viewed as an ideal communication channel which has no delay for any event data types (unlimited bandwidth). A process specified in the synchronous paradigm is a synchronous process. For feedback loops, the perfect synchrony creates cyclic dependency between output and input, and thus leads to deadlock, which can be resolved with initial events in the specification. A synchronous model is deterministic, i.e., given the same input streams, it generates the same output streams.

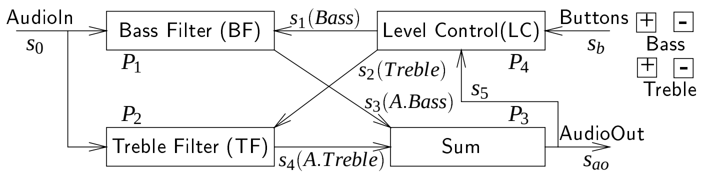


Figure 1.2. The digital equalizer

$$\begin{array}{lcl}
AudioOut & = & Equalizer(Buttons, AudioIn) \\
\text{where} & & \\
AudioOut & = & Sum(AudioBass, AudioTreble) \\
(Bass, Treble) & = & LevelControl(Buttons, AudioOut) \\
AudioBass & = & BassFilter(AudioIn, init : Bass) \\
AudioTreble & = & TrebleFilter(AudioIn, init : Treble) \\
init & = & 1
\end{array}$$

As a tutorial example, Fig. 1.2 illustrates an equalizer model. It adjusts the bass and treble volume of the audio stream according to button control levels. In addition it prevents the bass level from exceeding a predefined threshold to avoid damaging the speakers. Its function can be described by the set of equations, where the initial value '1' is used to resolve the feedback loops. This model is specified in the functional language Haskell and is executable.

3.2 Nostrum communication services

In Nostrum, each resource R_i ($i = 1, 2, \cdots, n$) is equipped with a Resource-Network-Interface (RNI) in order to access the network, as shown in the lower part of Fig. 1.3. The RNI and the network belong to the Nostrum protocol stack. Nostrum provides a message passing platform with two communication services, i.e., best-effort and guaranteed bandwidth. The BE service [12] is connection-less. Packets are routed in the network without reserving network resources such as storage and link bandwidth. The end-to-end flow control, re-ordering, packetization and packet admission control are performed by RNIs. The BE service maintains message order, and is lossless and corruptless. It has no guarantee on timely delivery, but must have an upper bound on delivery time. To this end, we assume that the communication protocols can prevent the network from saturation and guarantee bounds on delay. The GB service is connection-oriented. Bandwidth is negotiated during a connection establishment phase. Packets are delivered after a connection is established. The GB service is implemented by using looped containers and temporally disjoint networks [11]. The RNIs hide the service implementation details and make the services *transparently* accessible to applications. The access methods are communication primitives offered to the higher layer.

Within Nostrum, we define a set of basic communication primitives for message passing as follows:

- *int open(int src, int dst, int service, struct bandwidth)*: it opens a simplex channel between a source *src* process and a destination *dst* process. The *service* denotes the channel service class, 0 for the BE service, 1 for the GB service. The *bandwidth* is a user-defined record with three fields {*int min_bw, int avg_bw, int max_bw*} which specifies the minimum, average and maximum bandwidth (*Bytes*/*second*) requirement of the channel. The method returns a unique channel identity number (*cid*) upon successfully opening the channel; otherwise, it returns various reasons of failure, such as a destination invalid, or performance not satisfied.

- *bool write(int cid, void msg)*: it writes *msg* to the specified channel *cid*. The size of messages is bounded. It returns the status of the write.

- *bool read(int cid, void *msg)*: it reads channel *cid* and writes the received data to the address starting at *msg*. It returns the status of the read.

We have implemented these primitives with the BE service using SystemC in our layered NoC simulator *Semla* [13]. The write() and read() are presently implemented with nonblocking semantics. Semla is programmable as to network topology, process-to-resource mapping, routing algorithm, and traffic pattern. The current implementation opens channels statically during compile time and the opened channels are never closed during simulation.

3.3 The refinement procedure

Given a synchronous system specification, our objective is to refine the synchronous communication onto the Nostrum best-effort (BE) service. For this communication refinement, we propose a three-step procedure: *channel refinement*, *process refinement*, and *communication mapping*. We illustrate the procedure via a pair of producer-consumer processes in Fig. 1.3. The three steps are marked by a circle with a step number inside it.

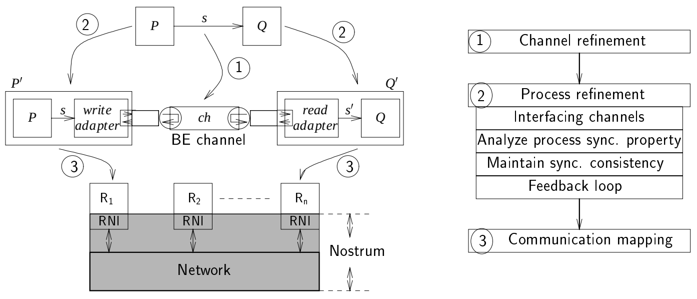


Figure 1.3. Communication refinement overview

Step 1: With **channel refinement**, we first abstract the behavior of the Nostrum BE service as that of stochastic channels which are then used to replace the ideal communication channels for passing signals. In Fig. 1.3, the ideal channel for signal s between producer P and consumer Q is refined to a BE service channel *ch*. After being delivered via the service channel, signal s turns into signal s', which is a derived version of s. Furthermore, s and s' are not synchronous since different clock domains are involved in the service channel.

Step 2: With **process refinement**, we discuss how to connect a process to the service interface and how its synchronization property can be met by using adapters to wrap the process. Particularly, to guarantee a correct refinement, the process synchronization property must be consistent from the specification to the refined model. We classify and analyze the synchronization property of processes and then discuss how to maintain *synchronization consistency*. The process synchronization property can be annotated by designers on processes to enable automatically instantiating *synchronizers* to achieve synchronization consistency in the process refinement. Moreover, we consider design decisions to handle feedback loops by which the process synchronization may be relaxed in order to optimize performance since a synchronous specification may overspecify the system. In Fig. 1.3, P and Q are wrapped with a write and a read adapter, respectively. Note that an adapter contains both a component to interface with the service channel (writer/reader) and component(s) to achieve synchronization consistency (synchronizers) whenever necessary.

Step 3: Finally, together with a process-to-resource allocation scheme, the **communication mapping** is to implement the adapters and map the service channels on a NoC, in this case, the Nostrum simulator Semla. In Fig. 1.3, the refined processes P' and Q' are mapped to the resources R_1 and R_n, respectively. Accordingly, the service channel *ch* is implemented via the interfaces provided by the RNIs of the resources R_1 and R_n.

4. Channel Refinement

The Nostrum BE service provides in-order, lossless and bounded-in-time communication between processes. However, its performance is *nondeterministic* since the message delivery experiences dynamic contentions in the RNIs and network. To capture the characteristics of the BE service, we resort to a stochastic approach. Formally, we develop a unicast BE service channel as a point-to-point *stochastic* channel: given an input signal of messages $\{m_1, m_2, \cdots, m_n\}$ to the service channel, the output signal is $\{d_1, m_1, d_2, m_2, \cdots, d_n, m_n\}$, where message m_i $(i = 1, 2, \cdots, n)$ is bounded in size; d_i denotes the delay of m_i which may be expressed as the number of absent ($\sqcup$) values and is subject to a distribution with a minimum $d_{i,min}$ and maximum $d_{i,max}$ value. The actual distribution, which may differ from channel to channel, is irrelevant. We do not make any further assumptions about this. If $d_i = n$ (n is a positive integer), it means there are n absent values between m_{i-1} and m_i. We can identify two important properties of the generic service channel behavior: (1) d_i is varying; (2) d_i is bounded. This behavior is purely viewed from the perspective of application processes and its implementation details are hidden.

Replacing the ideal channel (zero delay and unlimited bandwidth) with a stochastic channel (varying delay and limited bandwidth) leads to the violation of the synchrony assumption. In the specification, a channel is ideal so that we can use a *single* signal s to connect a producer to a consumer process. After replacing the ideal channel with a service channel, the signal s can be seen as being *split* into a pair of signals, the original signal s and its derived signal s', as shown in Fig. 1.3. For a process with two synchronous input signals, for example, the *Sum* process of the equalizer (Fig. 1.2), if both signals s_3 and s_4 are delivered via a service channel, they are split, resulting in two derived signals s'_3 and s'_4, which are now the input signals to the *Sum* process. Apparently, the two pairs of signals, s_3 and s'_3, s_4 and s'_4, and the two derived signals s'_3 and s'_4 are not synchronous. A synchronous system becomes globally asynchronous, leading to possibly nondeterministic behavior which deviates from the specification. It is therefore important for a refinement to maintain synchronization consistency for functional correctness.

5. Process Refinement

We first describe how to interface with the service channels in general, and then discuss the synchronization property of processes followed by methods to achieve synchronization consistency. At the system level (a composition of processes), we discuss feedback loops.

5.1 Interfacing with the service channels

Once an ideal channel is replaced by a service channel, the processes can not be directly connected to the interface of the service channel. They must be *adapted* in terms of data and control because (1) the input/output data type of a service channel is of a bounded size while a signal in the specification assumes an ideal data type, whose length is finite but arbitrary, e.g., a 32/64-bit integer, a 64-bit floating point or a user-defined 256-bit record type etc.; (2) the service channel has bounded buffers and limited bandwidth while a signal uses unlimited resources. The sending and receiving of messages use shared resources and thus control functionality has to be added to allocate shared resources, schedule multiple threads and achieve thread-level synchronization. These adaptations are achieved by a writer and reader process. Specifically, to interface with the service channels, a producer needs to be wrapped with a *writer*, a consumer with a *reader*.

5.2 Process synchronization property

In the system model, all signals of processes are synchronous. However, whether or not the input signals of a process must be synchronous is subject to the evaluation condition of processes, specifically, the local condition(s) to *evaluate* the input events. Because of the tight synchronization in the model, some processes may be over specified, limiting the implementation alternatives. During the refinement, the designer(s) must inspect and determine the synchronization property of the processes.

Inspired by [8], we use *firing rules* to discuss the synchronization property of *synchronous processes*. For a synchronous process with n input signals, PI is a set of N input patterns, $PI = \{I_1, I_2, \cdots, I_N\}$. The input patterns of a synchronous process describe its firing rules, which give the conditions of evaluating input events at each event cycle. I_i ($i \in [1,N]$) constitutes a set of event patterns, one for each of n input signals, $I_i = \{I_{i,1}, I_{i,2}, \cdots, I_{i,n}\}$. A pattern $I_{i,j}$ contains only one element that can be either a token wildcard $*$ or an absent value $\sqcup$, where $*$ does not include $\sqcup$. Based on the definition of firing rules, we propose four levels of process synchronization properties as follows:

- *Strict synchronization.* All the input events of a process must be present before the process evaluates and consumes them. The only rule that the process can fire is $PI = \{I_1\}$ where $I_1 = \{[*], [*], \cdots, [*]\}$.

- *Nonstrict synchronization.* Not all the input events of a process are absent before the process fires. The process can *not* fire with the pattern $I = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$.

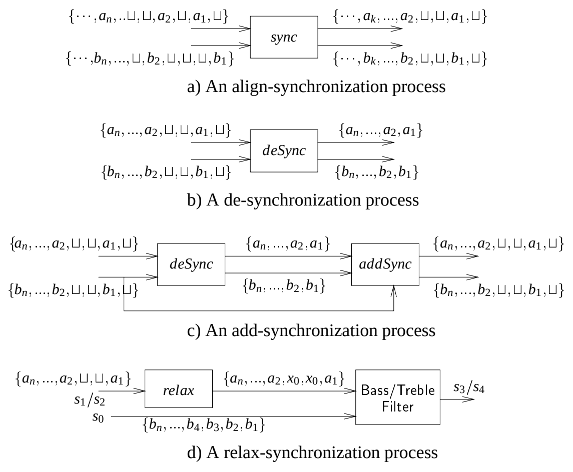
- *Strong synchronization.* All the input events of a process must be either present or absent in order to fire the process. The process has only two firing rules $PI = \{I_1, I_2\}$, where $I_1 = \{[*], [*], \cdots, [*]\}$ and $I_2 = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$.

- *Weak synchronization.* The process can fire with any possible input patterns. For a 2-input process, its firing rules are $PI = \{I_1, I_2, I_3, I_4\}$ where $I_1 = \{[*], [*]\}$, $I_2 = \{[\sqcup], [\sqcup]\}$, $I_3 = \{[*], [\sqcup]\}$ and $I_4 = \{[\sqcup], [*]\}$.

We can identify processes with a *strict*, *strong*, and *weak* synchronization property in the equalizer (Fig. 1.2). The $BassFilter$ (s_0 and s_1) and $TrebleFilter$ (s_0 and s_2) have a strict synchronization. Both filters are composed of a FIR filter and an amplifier. The FIR filter is specified as an FSM, whose state transition is sensitive to time, thus a $\sqcup$ value in an audio stream can change the values of its output sequence. Meanwhile, the amplifier must have an amplification level, thus a $\sqcup$ value makes the amplifier undefined. The *Sum* process (s_3 and s_4) has a strong synchronization. It is a combinational process and thus tolerable to events with a $\sqcup$ value. However, the two events of s_3 and s_4 must be synchronized before being processed since they represent the low and high frequency components of the same audio sample. The *LevelControl* (s_b and s_5) process has a weak synchronization. It can fire even when either or both of the events of s_b and s_5 are absent since pressing buttons happens irregularly and the bass level surpassing the threshold occurs only aperiodically.

5.3 Achieving synchronization consistency

Apparently, for processes with a strict or strong synchronization, their synchronization properties can not be satisfied if any of their input signals passes through a service channel since the delays via the channel are stochastic. Although globally asynchronous, the processes can be locally synchronized by using *synchronizers* to satisfy their synchronization properties. To achieve strong synchronization, we use an align-synchronization process *sync*; to achieve strict synchronization, we use three processes, *sync*, *deSync* and *addSync*. We use a two-input process to illustrate these processes in Fig. 1.4. An align-synchronization process *sync* aligns the tokens of its input events, as shown in Fig. 1.4a. It does not change the time structure of the input signals. A desynchronizer *deSync* removes the absent values, as shown in Fig. 1.4b. All its input signals must have the same token pattern, resembling the output signals of the *sync* process. Removing absent values implies that the process is *stalled*. The desynchronizer changes the timing structure of the input signals, which must be recovered in order to prevent from causing unexpected behavior of other processes that use the timing information. An add-synchronizer *addSync* adds the absent values to recover the timing structure, as shown in Fig. 1.4c. It must be used in relation to a *deSync* process. If the input events of the *deSync* is a token, the *addSync* reads one event from its internal buffers for each output signal; otherwise, it outputs a ⊔ event. The two processes *deSync* and *addSync* are used as a pair to assist processes to fulfill strictness.



$\{\cdots, a_n, ..\sqcup, \sqcup, a_2, \sqcup, a_1, \sqcup\}$
$\{\cdots, b_n, ..., \sqcup, b_2, \sqcup, \sqcup, \sqcup, b_1\}$
sync
$\{\cdots, a_k, ..., a_2, \sqcup, \sqcup, a_1, \sqcup\}$
$\{\cdots, b_k, ..., b_2, \sqcup, \sqcup, b_1, \sqcup\}$
a) An align-synchronization process
$\{a_n, ..., a_2, \sqcup, \sqcup, a_1, \sqcup\}$
$\{b_n, ..., b_2, \sqcup, \sqcup, b_1, \sqcup\}$
deSync
$\{a_n, ..., a_2, a_1\}$
$\{b_n, ..., b_2, b_1\}$
b) A de-synchronization process
$\{a_n, ..., a_2, \sqcup, \sqcup, a_1, \sqcup\}$
$\{b_n, ..., b_2, \sqcup, \sqcup, b_1, \sqcup\}$
deSync
$\{a_n, ..., a_2, a_1\}$
$\{b_n, ..., b_2, b_1\}$
addSync
$\{a_n, ..., a_2, \sqcup, \sqcup, a_1, \sqcup\}$
$\{b_n, ..., b_2, \sqcup, \sqcup, b_1, \sqcup\}$
c) An add-synchronization process
$\{a_n, ..., a_2, \sqcup, \sqcup, a_1\}$
s_1/s_2
s_0
relax
$\{a_n, ..., a_2, x_0, x_0, a_1\}$
$\{b_n, ..., b_4, b_3, b_2, b_1\}$
Bass/Treble Filter
s_3/s_4
d) A relax-synchronization process

Figure 1.4. Processes for synchronization

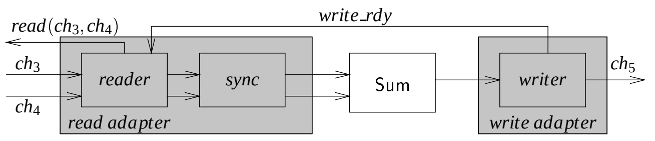


Figure 1.5. Read/Write adapters for a process with strong synchronization

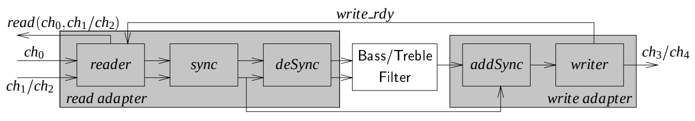


Figure 1.6. Read/Write adapters for a process with strict synchronization

We can now use these synchronizers in connection with the *reader* and *writer* processes to wrap the original processes to interface with the service channels and maintain the synchronization consistency from the specification model to the refined model. For instance, as shown in Figure 1.5, we use a *sync* process and a pair of *reader*/*writer* processes to wrap the *sum* process in the equalizer to maintain its strong synchronization. We use the three processes, *sync*, *deSync* and *addSync*, and a pair of *reader*/*writer* processes to wrap the *Bass*/*Treble Filter* process (Fig. 1.2) to maintain their strict synchronization.

The refinement of processes with a nonstrict synchronization should be individually investigated according to their firing rules.

5.4 Feedback loops

In the specification, feedback loops are resolved by using initial events. If the feedback signals pass through a service channel, the delays are nondeterministic. If following the initial event approach in the refinement procedure, we encounter a problem since we are not certain how many initial events are required to resolve the deadlock. Consider the *Bass*/*Treble Filter*, if the tokens of s_1/s_2 are not available, it can not fire. This implies it may not be able to process enough audio samples in time, leading to violate the system's performance constraint. However, if the amplification level signals, s_1 (*Bass*) and s_2 (*Treble*), are delayed and thus not available, the amplifiers should continue functioning by, for example, using the previous amplification level or simply using a constant level like 1. In this case, the effect of pressing buttons may be delayed several cycles. This is tolerable since the human sensing of the changes in the audio volume is not instantaneous.

By this observation, we can in fact *relax* the strict synchronization of the processes *Bass*/*Treble Filter*, using a relax-synchronization process *relax* illustrated in Fig. 1.4d. If the input event is a token, it outputs the token; otherwise, a token x_0 is emitted. The exact value of x_0 is application dependent. Relaxing synchronization is a design decision leading to behavior discrepancy between the specification and the refined model. Care must be taken to validate the resulting system.

6. Communication Mapping

The inputs to this task are the refined model as well as a process-to-resource allocation scheme; the output is a communication implementation on Semla.

6.1 Channel mapping

With a resource allocation scheme, all processes are allocated to resources in a one-to-one manner. Note that this is not a limitation but due to the assumption on the clustering and resources (refer to Section 1). With such a clustering, inter-process signals, which represent inter-resource communications, are mapped to service channels. Since the processes may be hierarchical, we need to flatten the hierarchy to the level that each signal mapped to a service channel can be uniquely identified with a pair of a producer and a consumer process with *finer* granularity. For simplicity, we do not consider mapping multiple service channels to one implementation channel. Mapping channels is thus straightforward. Each pair of processes communicating via a service channel in the refined model results in its dedicated unicast implementation channel, which is mapped to the open channel primitive *open()*. For example, with the producer-consumer case, a BE channel setup is fulfilled by a single line of code: $int\ ch[1] = open(P, Q, BE\_SERVICE, NULL)$.

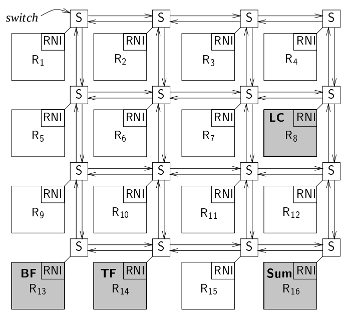
6.2 Communication process mapping

After the process refinement, a refined process consists of the original computational process, the writer and reader, and perhaps the synchronizer(s) to satisfy their synchronization properties. Our refinement keeps the original processes intact. Therefore, the tasks of communication process mapping are to implement the writer/reader, and the synchronizers such as *sync*, *deSync*, *addSync* and *relax*, and to coordinate the writing and reading operations.

In SystemC, processes are implemented as modules. The reader/writer may be implemented as separate modules or in the same modules as processes. We implement a process and its adapter(s) in a single module. For implementation, execution control in the module must be considered. Suppose the module has a single thread of control, we need to find a Periodic Admissible Sequential Sequence (PASS) for process executions [10]. For the process in Fig. 1.6, a PASS could be PASS={*reader*, *sync*, *desync*, *compute*, *addsync*, *writer*}. Besides, a control signal *write_rdy* must be asserted by the *writer* to the *reader* to enable reading the channel(s) for the next-round PASS execution, as shown in Fig. 1.6. This leads to a local feedback loop, and we adopt the initial event approach to deal with it. In this case, *write_rdy* is initially asserted. Using the communication primitives defined in Section 3.2, the SystemC module for Fig. 1.6 is sketched as follows, with each component explained in commentary:

```
process_class::Process(){
  //initially  write_rdy=1;
  //read_ch0_rdy=0; read_ch1_rdy=0
  //sync_rdy=0; compute_done=0;
  if (write_rdy==1){
  //(1) reader: nonblocking read ch1 and ch2
   if (read_ch0_rdy==0)
     if ((read(ch[0],&r_msg1))==true)
        read_ch0_rdy=1;
   if (read_ch1_rdy==0)
     if ((read(ch[1],&r_msg2))==true)
        read_ch1_rdy=1;
  //(2) sync: synchronize the two events
  if (read_ch0_rdy==1 && read_ch1_rdy==1)
     sync_rdy=1;
  else sync_rdy=0;
  //(3) deSync: desynchronization by guard
  if (sync_rdy==1 && compute_done==0){
     //process computation
     //return w_msg and set compute_done to 1
     w_msg=compute(r_msg1, r_msg2);
     write_rdy=0; compute_done=1;}
 }
 //(4) addSync: fill synchronization
  if (sync_rdy==1 && compute_done==1) {
  //(5) writer: nonblocking write ch3
   if (write_rdy==0)
     if (write(ch[3],w_msg)==true){
        write_rdy=1;
        sync_rdy=0; compute_done=0;
        read_ch0_rdy=0; read_ch1_rdy=0;}}
}
```

In the implementation domain, whether to emit and pass $\sqcup$ either as a special message or using one bit to indicate *presence* and *absence* via a service channel can be a design decision. To preserve the semantics, $\sqcup$ must be transported. However, this incurs too much overhead on computation and communication, and may be meaningless since its value is useless. Therefore $\sqcup$ is usually neglected. Only in cases where the timing information carried by $\sqcup$ is used by other processes, it must be emitted and passed. In the equalizer case, $\sqcup$ is neglected since its timing information is not used by any of the four processes.



(a) Resource allocation

Load (%)	Avg. Delay (cycles)	Throughput (samples/cycle)
9.06	15.00	0.0667
16.96	16.53	0.0605
25.16	18.52	0.0540
29.97	20.24	0.0494
35.36	22.59	0.0443
43.28	35.14	0.0285

(b) Performance

Figure 1.7. The equalizer mapped on a NoC

We have implemented the equalizer in Semla. The purpose is to validate the concepts of our refinement approach. Fig. 1.7a illustrates the mapped equalizer in a 4x4 mesh NoC. All the five inter-resource signals $s_1, s_2, \cdots, s_5$ (Fig.

1.2) use the BE service. The resources and the network run with the same speed. The switches operate synchronously with the switching per hop taking one cycle. The message streams on s_3 and s_4 are injected into the network conservatively so that a new audio sample will not be processed by the filters until the previous sample has been handled by the *Sum* process. This implies that the audio samples are not processed in a pipeline fashion in the network. In addition, we inject background traffic with uniformly distributed random destinations in the network. The motivation is to load the network with reasonable amount of traffic since the equalizer example can only make use of a small fraction of the network capacity. Fig. 1.7b shows the equalizer performance, where the network load is the average percentage of active links per cycle. The process computations are function calls and complete instantly. We observe the average delay that is the time (in cycles) to process one sample. Since the audio processing is not pipelined, the throughput (samples/cycle) is simply the inverse of the average delay. In Fig. 1.7b, the first row shows the case where there is no background traffic. As expected, when the network is increasingly loaded, the average delay is increased and the throughput decreased. The average delay can be seen as the time to respond to a button press or to activate bass control. We noted that the audio output sequences are different from those observed from the specification due to relaxing the synchronization for the feedback loops. We conducted other experiments in which we removed the feedback loops, and could validate that the output sequences agree with each other in all traffic setting cases.

7. Conclusions and Future Work

Communication refinement is a crucial step in a NoC design flow. We have presented a refinement approach that allows us to map a perfectly synchronous communication model onto the NoC best-effort service accessible through communication primitives. Particularly we classify the synchronization properties of processes and describe methods to achieve synchronization consistency during the refinement upon the violation of the perfect synchrony hypothesis. For feedback loops, we relax the synchronization with the tolerance of system requirements. In this paper we use Nostrum as target, but with few adjustments, this approach is also applicable for other NoC platforms.

In future work, we plan to develop formalism for synchronization consistency and realize automatically analyzing the synchronization properties of processes. During refinement, we take either automatic analysis that yields correct synchronization and system behavior, or manual analysis with design decisions on the synchronization refinement combined with a systematic verification of the resulting implementation. For the refinement of feedback loops, we intend to use the Nostrum GB service to reach a systematic solution.

References

[1] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation,* 163:125–171, 2000.

[2] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *Proceedings of the Third International Conference on Embedded Software,* 2003.

[3] J.-Y. Brunel, W. Kruijtzer, H. Kenter, F. Petrot, L. Pasquier, E. de Kock, and W. Smits. COSY communication IP's. In *Proceedings of the 37th Design Automation Conference*, Los Angeles, California, June 2000.

[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sept. 2001.

[5] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of Design Automation and Test in Europe,* 2003.

[6] R. Dömer, D. D. Gajski, and A. Gerstlauer. SpecC methodology for high-level modeling. In *Proceedings of the Ninth IEEE/DATC Electronic Design Processes Workshop,* April 2002.

[7] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–303, December 2003.

[8] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 1995.

[9] C. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of Design Automation and Test in Europe*, 2000.

[10] Z. Lu, I. Sander, and A. Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, October 2002.

[11] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Design Automation and Test Europe Conference (DATE)*, 2004.

[12] E. Nilsson, M. Millberg, J. Öberg, and A. Jantsch. Load distribution with the proximity congestion awareness in a network on chip. In *Proceedings of the Design Automation and Test Europe (DATE),* pages 1126–1127, 2003.

[13] R. Thid, M. Millberg, and A. Jantsch. Evaluating NoC communication backbones with simulation. In *Proceedings of the IEEE NorChip Conference*, 2003.