Towards Performance-oriented Pattern-based Refinement of Synchronous Models onto NoC Communication

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 performance-oriented refinement approach that refines a perfectly synchronous communication model onto Network-on-Chip (NoC) communication. We first identify four basic forms of NoC process interaction patterns at the process level, namely, producer-consumer, peers, client-server, and multicast. We propose a three-step top-down refinement method: channel refinement, protocol refinement and channel mapping. For the producer-consumer pattern, we describe it in detail. In channel refinement, we deal with interfacing multiple clock domains and use a stochastic process to model channel delay and jitter. In protocol refinement, we show how to refine communication towards application requirements such as reliability and throughput. In channel mapping, we discuss channel convergence and channel merge arising from channel overlapping. All the refinements have been conducted and validated as an integral design phase towards implementation in ForSyDe, a formal system-level design methodology based on a synchronous model of computation.

1 Introduction

Network-on-Chip (NoC) is deemed to be a paradigm to tackle System-on-Chip (SoC) design challenges in the billion transistor era. Due to lack of scalability, a bus-based (single level or multi-level) architecture is becoming performance bottleneck to interconnect tens or even hundreds of microprocessor-sized heterogeneous resources. Most probably a bus-based design will be used at the local resource level and complemented with a network platform at the chip global level. Meanwhile the deep submicron technology limits the maximum synchronous region on a chip to a local resource area. Globally Asynchrony Locally Synchrony (GALS) is regarded as a future SoC synchronization mechanism. From the design methodology perspective, raising design abstraction to system level is considered to be indispensable to cope with relentlessly increasing design complexity.

Keutzer et al. discuss system-level design in [12]. They point out, that "to be effective a design methodology that addresses complex systems must start at high levels of abstraction". Particularly, a design methodology should separate (1) function (what the system is supposed to do) from architecture (how it does it) and (2) communication from computation. They "promote to use formal models and transformations in system design so that verification and synthesis can be applied to the advantage of the design methodology". These arguments not only support but also establish the foundations of ForSyDe. The ForSyDe [13, 14] methodology addresses the design of SoC applications. Starting with a formal system specification model that captures the system functionality at a high abstraction level, it provides formal transformation methods to refine the system model into an implementation model , which serves as a starting point for synthesis into HW and SW.

In this paper we present the top-down communication refinement method towards NoC communication in ForSyDe, focusing on techniques to satisfy communication reliability and to leverage throughput and network utilization. The related work is briefed in section 2. We then introduce the ForSyDe methodology in section 3, and the process communication patterns, our NoC platform and its services in section 4. In section 5, we discuss our incremental communication refinement steps, *channel refinement, protocol refinement* and *channel mapping*. A tutorial example is shown in section 6, followed by conclusion in section 7.

2 Related Work

Based on the isolation of communication from computation, a large body of work on communication refinement exists in the literature. Through the Virtual Component Interfaces (VCI) of the VSI Alliance [10], 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]. In the course of communication refinement, methods to allow architecture exploration and protocol selection can be found in [11] and [9], respectively. These works do not assume a synchronous specification, thus are not applicable to our context.

With synchronous communication, latency insensitive theory [4] targets synchronized HW design where synchronization can still be achieved even if interconnecting synchronous IP blocks experiences indefinite wire latencies; De-synchronization 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 theoretical framework is proposed in [8] concerning the refinement of a polysynchronous specification, which allows multiple clocks instead of a single clock. All these works are complementary to our work but none of them provides a detailed refinement approach targeting a NoC platform. Furthermore, this paper concentrates on refinement techniques to satisfy performance requirements based on process interaction patterns.

3 The ForSyDe Methodology

3.1 The Design Process

The ForSyDe [13, 14] design process starts with the development of an abstract functional specification expressed in the functional language Haskell. This model is then refined inside the functional domain by a stepwise application of well defined design transformations into an efficient implementation model. As the implementation model is a refined version of the specification model, the same validation and verification methods can be applied to both models. In the partitioning phase, the implementation model is partitioned into HW and SW blocks, which are mapped on architectural components. Only now, in the code generation phase, we leave the functional domain to generate VHDL or C code for the HW and SW.

3.2 The Specification Model

The specification model follows the synchronous modeling paradigm. This 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. In order to formally describe our synchronous computational model, we follow the denotational framework of Lee and Sangiovanni-Vincentelli [16]. They define signals as a set of events, where each event e has a tag t and a value v, i.e. $e = (t, v) \in T \times V$. As our system model is synchronous, T is the set of natural numbers, and all signals have the same set of tags. In order to model the absence of an event, a data type D can be extended into a data type $D_\perp$ by adding the special value $\perp$, which is used to model the absence of a value. Absent values are used to establish a total order of events when dealing with signals with different or aperiodic event rates.

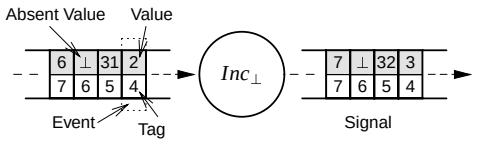


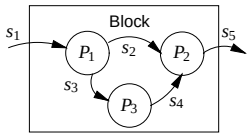
Figure 1: Modeling of signals and processes

Figure 1 illustrates the modelling of signals and the behavior of processes. At the event cycle n a process evaluates the events of each signal with the tag n and outputs the result at the same tag n.

We implement the synchronous computational model with the concept of *process constructors*. A *process constructor* is a higher order function that takes *combinational functions*, i.e. functions that have no internal state, and *values* as input and produces a process as output. There is a clean separation between *synchronization* (captured by process constructors) and *computation* (implemented by combinational functions). In addition, each process constructor has a structural *hardware* and *software* semantics which is used to translate the implementation model into a hardware/software implementation.

As an example, the process constructor *mealySY* models a finite state machine of Mealy type. It takes a function *ns* to calculate the next state as first argument, a function *out* to calculate the output as second argument and a value s_0 for the initial state as last argument. Thus a process $Mealy = mealySY(ns, out, s_0)$ implements the behavior of a finite state machine.

Processes can be glued together to build a network of processes. Such a network is called a block. Figure 2 shows how a block is formed by a network of processes. The function of a block is expressed by a set of equations.



$$
\begin{array}{lll}
Block(s_1) = & s_5 & \\
\text{where} & (s_2, s_3) & = P_1(s_1) \\
& s_5 & = P_2(s_2, s_4) \\
& s_4 & = P_3(s_3)
\end{array}
$$

Figure 2: A network of processes

In the same way, blocks can be composed into higher level blocks, subsystems and eventually a hierarchical system.

3.3 Refinement of the Specification Model

The specification model abstracts from implementation details, such as buffer sizes and low-level communication mechanisms. This enables the designer to focus on the functional behavior of the system rather than structure and architecture. This abstract nature leaves a wide space for further design exploration and design refinement, which is supported by our transformational refinement technique. During the refinement phase the specification model is stepwise refined into a final optimized implementation model.

4 NoC Communication

4.1 Process Communication Patterns

A NoC application can be represented as a process network with a set of functional equations in ForSyDe. According to the interactions among processes [7], we identify the basic forms of inter-process communication patterns as follows:

- Producer-consumer. This is a one-to-one pattern where a producer generates data which in turn are consumed by a consumer. The consumer may or may not send back acknowledgments depending on application requirements.
- Peers. Similar to the producer-consumer, but both processes send/receive data and perhaps acknowledgments to/from each other.
- Client-server. This is a multiple-to-one pattern. Multiple clients send requests to a server which responds with various services. It works in a request-response manner in that a server will not respond to a client until a valid request is asking for service. The server may offer a uniform service or multiple services. For example, a memory only serves data read/write service. A microprocessor may provide various computing services such as remote procedure calls.
- Multicast. This is a one-to-multiple pattern. The group master actively reads/writes data to its multicast group members.

Next, we will take the producer-consumer pattern to demonstrate our communication refinement approach.

4.2 The NoC Platform and its Services

Our NoC platform [15] is a mesh structure composed of switches where each switch is connected to a resource, as shown in figure 3. The resources, which may work with different clock rates, are placed on the slots formed by the switches. The area of a resource is constrained within the maximal synchronous region in a given technology. The Resource-Network-Interfaces (RNIs) offer network communication services to resources.

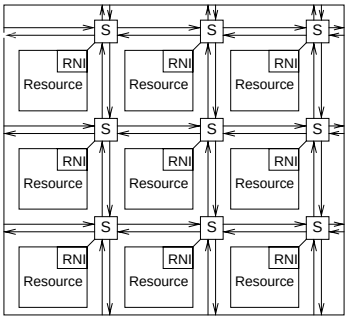


Figure 3: A NoC of mesh structure with 9 nodes

Our NoC platform provides two kinds of services. One is *best-effort* or connection-less delivery of messages. The other is connection-oriented *virtual circuit* that provides a resource-to-resource connectivity. In best-effort service, messages are routed in the network. The data sequence is maintained by re-ordering and data will not be lost. Neither the bandwidth nor the latency can be guaranteed. In virtual circuit services, messages are reliably delivered in order with guaranteed bandwidth. In terms of latency predictability, the services can be further classified as two kinds: *virtual circuit with latency commitment* and *virtual circuit with relaxed latency commitment*. In both cases, there is a bounded range with minimum and maximum latency value. For the second case, the upper bound is the worst case latency along the virtual circuit path.

Our NoC architecture provides a message passing platform. Processes communicate in the platform via channels which use one of the services. A message passing procedure between processes consists of three phases:

channel setup, data transmission and channel tear-down. If a channel intends to use the best-effort service, the channel setup is handled locally relying on an admission protocol in order not to saturate the network. If a channel asks for a virtual circuit service, the initiating process sends a setup message using the best-effort service to negotiate with the network for bandwidth and delay during the channel setup phase. Once the request is granted, the circuit path is fixed, and the bandwidth is reserved. The data transmission unit from process to process is message. A message comprises a channel identity number and payload. After the data transmission phase, the channel has to be explicitly torn down.

5 NoC Communication Refinement

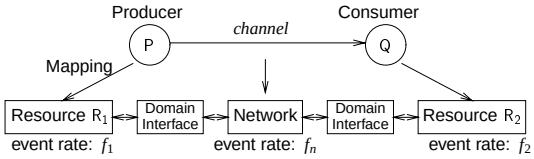
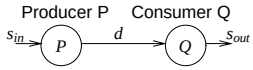


Figure 4: The producer-consumer and the target NoC

In figure 4, the upper part shows a producer-consumer pattern where the producer P communicates with the consumer Q via a *logical channel* [1]. The lower part shows a NoC instance with two resources R_1, R_2. In a GALS architecture, the resources R_1, R_2 and the network may work in different clock domains f_1, f_2 and f_n, respectively. We assume that all clocks have the same phase. The time structures from these clock domains have to be arbitrated when cross-domain communication is incurred. To this end we have explicitly highlighted the two domain interfaces, though they may be implemented as part of RNIs on resources. Our task is to map the producer-consumer onto the NoC. As for this pattern, the fundamental problem is data loss which occurs when the data-producing speed is higher than the data-consuming speed. Our refinement is not solving this problem. Instead we assume that the data-producing speed is not higher than the data-consuming speed. Also, the only meaningful read scheme for the consumer in this case is blocking read since the consumer can not react if no data is received.

In ForSyDe this pattern is initially modeled with a network of two processes, P and Q, as shown in figure 5. Process P models the producer P. Process Q models the consumer Q. The two processes communicate in a perfectly synchronous manner via the signal d. A *signal* in the specification is to be mapped to a *service channel*.

[1] For convenience, we also call an arc connecting a pair of interacting processes a *channel*. It is logical and not associated with a service yet.



$$\begin{aligned} & s_{out} = QP(s_{in}) \\ \text{where} \quad & d = P(s_{in}) \\ & s_{out} = Q(d) \end{aligned}$$

Figure 5: The producer-consumer model in ForSyDe

5.1 Refinement Overview

Our objective is to refine this perfectly synchronous producer-consumer model onto the NoC communication services. The service selection is subject to the channel characteristics. The resultant producer and consumer model should fulfill application requirements such as reliability and throughput. For reliability, the producer asks from the consumer for acknowledgment for each message sent. For throughput, the producer-consumer has to make full use of the channel bandwidth honored during the channel setup. If the channel is not established, nothing will happen. Therefore we concentrate our refinement on the data transmission phase.

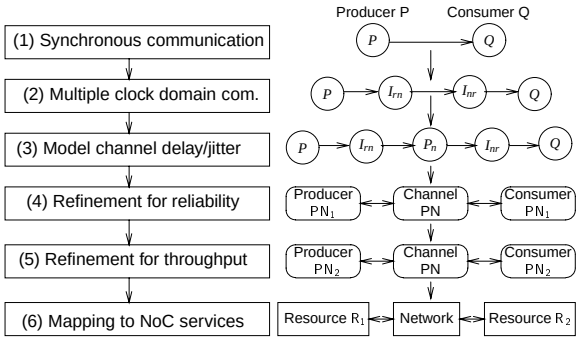


Figure 6: Refinement to NoC communication services

From now on we assume that the channel in figure 4 is granted with either of the two virtual circuit services. During the refinement steps we focus on how the process networks (PNs) will evolve. The overall refinement steps and the resultant process networks are illustrated in figure 6. The initial model (figure 5) is the perfectly synchronous model where there is only one clock domain. In step (2) we consider different clock domains and interfacing the clock domains. The process I_{rn} models the domain interface connecting a resource to the network. The process I_{nr} models the domain interface connecting the network to a resource. In step (3) we model the channel delay and jitter with the process P_n. We call the steps (2) and (3) *channel refinement* covered in section 5.2. In fact the channel refinement builds the channel model for refining the producer P and the consumer Q. In steps (4) and (5)

the process networks are refined to satisfy the reliability and throughput, respectively. We call the steps (4) and (5) *protocol refinement* covered in section 5.3. In step (6) covered in section 5.4 we discuss channel convergence and channel merge while mapping the channel to NoC communication services.

5.2 Channel Refinement

5.2.1 The clock domain interfaces

First of all we build models for the two clock domain interface processes I_{rn} and I_{nr}. Introducing a synchronous sub-domain into the system model was presented in [13] where the clock rate of the sub-domain is $\frac{1}{n}$ (n is a positive integer) of the main domain. Here we consider a generic domain interface that connects a clock domain with event/clock rate f_1 to another clock domain with event rate f_2. The simplest form of the fraction $\frac{f_1}{f_2}$ is $\frac{m}{n}$. The generic interface is constructed as $I_{f_1 \to f_2} = P_{dn}(m) \circ P_{up}(n)$, where $\circ$ is the composition operator. The processes, $P_{up}(n)$ and $P_{dn}(m)$, are formally defined as follows:

$$P_{up}(n)(\{x_1, x_2, \ldots\}) = \{\underbrace{\perp, \ldots, \perp}_{n-1}, x_1, \underbrace{\perp, \ldots, \perp}_{n-1}, x_2, \ldots\}$$

$$P_{dn}(m)(\{\underbrace{x_1, x_2, \ldots, x_m}_{m}, \underbrace{x_{m+1}, \perp, \ldots, \perp}_{m}, \ldots\}) = \{x_m, x_{m+1}, \ldots\}$$

The *up-sampling* process $P_{up}(n)$ samples out n times of the input events, and does not result in event loss. The *down-sampling* process $P_{dn}(m)$ samples out $\frac{1}{m}$ times of the input events. At each down-sampling cycle, $m-1$ events are discarded and only the last valid event value (non-absent value) is kept. The interface first does up-sampling and then down-sampling. If $f_1 \leq f_2$, no event drop, hence no data is lost. If $f_1 > f_2$, events are cyclically dropped. But data may or may not be lost because the data rate may not match the event rate. If there is no data at an event cycle, only an event with absent value $\perp$ is inserted into the signal.

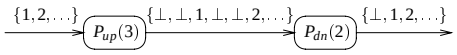


Figure 7: A clock domain interface

Figure 7 shows the interface process network for connecting the clock domain f_1 to the clock domain f_2 with the ratio $\frac{f_1}{f_2} = \frac{2}{3}$. Knowing the clock rates of the resources and the network, we can similarly build the interface processes I_{rn} and I_{nr}. Our assumption is that the data-producing speed is not higher than the data-consuming speed. Besides, the NoC communication services guarantee that no data will be lost at the network. The two conditions guarantee that there is no data loss at the interfaces I_{rn} and I_{nr}.

5.2.2 Model channel delay/jitter

We have assumed that the channel uses either of the two virtual circuit services fulfilling the bandwidth requirement. If viewing from a process's perspective, the net effect of delivering messages is delay and delay variances called jitter. To model the channel delay/jitter, we introduce stochastic characteristics to the network process P_n. The stochastic process $D_{[min,max]}$ generates a random delay within a given range $[min, max]$ for each event. A delay is modeled as an event with the absent value $\perp$. Figure 8 shows a stochastic delay process with the jitter range $[0,3]$.

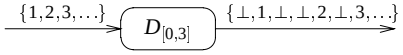


Figure 8: A stochastic delay process

After inserting the stochastic process, we receive a channel-refined producer-consumer, as shown in figure 9.

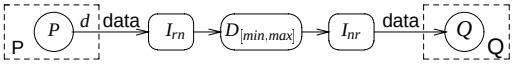


Figure 9: The channel-refined producer-consumer model

5.3 Protocol Refinement

5.3.1 The acknowledged producer-consumer

Although the channel is lossless and errorless, the consumer may be out of function or experience buffer overflow. In such a case, it is necessary for the producer to receive an acknowledgment before sending the next message in order to prevent the producer from overloading the network. This results in a feedback loop from the consumer to the producer shown in figure 10. If no acknowledgment is received, the producer will wait and not feed more data to the network, and the incoming data from the process P will be silently dropped.

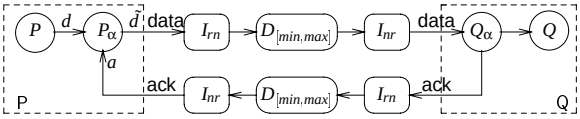


Figure 10: The acknowledged producer-consumer model

The processes P_α and Q_α in figure 10 implement the acknowledgment protocol. The process P_α has two states, *Idle* and *Wait*. It is modeled as a mealy FSM with the process constructor `mealySY` as follows:

$$\begin{array}{rcl} \tilde{d} & = & P_\alpha(d, a) \\ \text{where} \quad P_\alpha & = & mealySY(ns, out, Idle) \end{array}$$

The process Q_α receives data from the channel, then passes the data to the process Q and generates acknowledgment.

At this step, the producer P is refined into the two processes P and P_α. The consumer Q is refined into the two processes Q and Q_α. The reliability is achieved through acknowledgment.

5.3.2 Buffering

In figure 10, during waiting for acknowledgment, the next incoming data will be silently dropped. To avoid this, a bounded FIFO buffer process P_{buffer} is inserted between the process P and the process P_α. Data produced by the process P is first pushed into the buffer. The process P_α is refined into the process P_β that has an additional signal *readBuf* to read the buffer, as shown in figure 11. When the previously sent data is acknowledged, the process P_β reads the buffer until successfully fetching data.

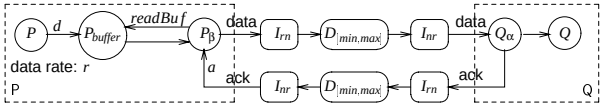


Figure 11: The acknowledged producer-consumer after buffering

It is easily seen that, if the data-emitting speed is higher than that of receiving acknowledgment, any bounded buffer will eventually overflow. The fastest data-emitting speed r_{max} without buffer overflow is governed by the following formula:

$$\exists\, r_{max}.\ r_{max} = \frac{f}{1 + 2 \cdot D_{min} \cdot f}$$

where D_{min} is the minimum channel delay and f the producer's clock frequency. The minimum buffer size is 1.

At this step, the producer P is refined into the three processes P, P_{buffer} and P_β shown in figure 11. The consumer Q has no change.

5.3.3 Data pipelining

In figures 10 and 11, each message is individually acknowledged. The data transmission speed is limited by the variable channel delay. This leads to a waste of channel bandwidth, and, in some cases, data loss at the producer due to the buffer overflow. To solve this problem we elaborate the protocol. Instead of generating one acknowledgment for one received message, we can acknowledge a batch of data altogether. After sending a batch of data with size w, the producer waits for an acknowledgment from the consumer. Upon receiving the acknowledgment for the w data, the producer starts to emit the next batch of data into the channel. In this way, the channel utilization is largely improved. The maximum allowable data-emitting speed without buffer overflow can be increased with a factor of nearly w, but no more than the channel capacity. The window size w is affected by the channel delay and bandwidth, and the consumer buffer size. It is initially determined during the channel setup phase. Later it may be dynamically adjusted in case of network congestion control. Further, we can even improve the channel throughput by *prediction*. We assume an acknowledgment will come at the right time, thus we can first emit $2 \times w$ size of data before waiting for the acknowledgment. If the acknowledgment for the first w data comes in time, the producer starts to emit the third batch, and so forth. Otherwise, the producer has to wait. Compared with the previous pipelining, this will improve the channel throughput by up to a factor of two leading to a fully data-pipelined channel.

To accomplish the data pipelining, we need a counter process at both the producer and the consumer side, shown in figure 12. The counter process $Q_{counter}$ at the consumer side counts up to the window size w and generates one acknowledgment. The counter process $P_{counterRst}$ at the producer side counts the number of sent data. If the window size w or $2w$ is not reached, more data can be fetched from the buffer. In contrast to the process $Q_{counter}$, the process $P_{counterRst}$ is reset upon receiving an acknowledgment. That means, if an acknowledgment comes, it restarts to count from 0.

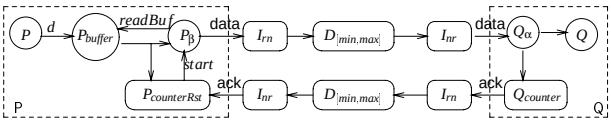


Figure 12: The acknowledged producer-consumer after windowing

At this step, the producer is refined into the four processes, and the consumer is refined into the three processes, shown in figure 12. Now the acknowledged producer-consumer can efficiently use the channel bandwidth. Our protocol refinement objective is thus achieved.

5.4 Channel Mapping

5.4.1 Channel convergence

After the protocol refinement, the producer and the consumer are mapped to their allocated resources. The channel uses the network communication services via the RNIs on the resources. Figure 13 shows two pairs of the non-acknowledged producer-consumer, P_1 and Q_1, P_2 and Q_2. The two producers P_1 and P_2 are mapped to the resource R_1. The two consumers Q_1 and Q_2 are mapped to the resource R_2. The two channels ch_1 and ch_2 use the virtual circuits vc_1 and vc_2 via RNI_1 and RNI_2, respectively.

One pre-condition for message passing is that the channel has to be established before communicating. Since no NoC platform can provide unlimited bandwidth, the number of channels which can be opened simultaneously is always limited. The channel setup may become the communication bottleneck. On the other hand, during the mapping of the producers and the consumers onto the NoC platform, some producers may be mapped to one resource and some consumers may be mapped to another resource, leading to overlapped channels. In some cases, if one virtual circuit can satisfy the latency requirement of these channels, and can provide enough bandwidth, the producers and the consumers can in fact share the virtual circuit. We call this *channel convergence*. In figure 13, if the latency of the virtual circuit vc_1 or vc_2 satisfies the latency requirements of the two channels, ch_1 and ch_2, and its bandwidth is not less than the sum of the two channel bandwidth, the two channels can share the virtual circuit vc_1 or vc_2.

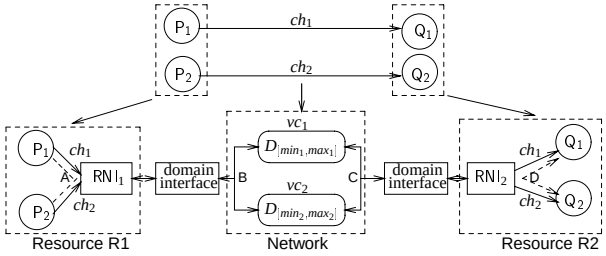


Figure 13: Channel mapping and channel convergence

5.4.2 Channel merge

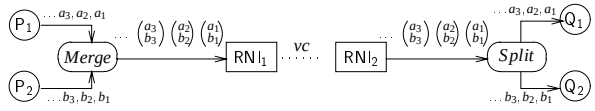


Figure 14: Channel merge and split

Further, if the message format can contain the payloads from the two channels, ch_1 and ch_2, there is a possibility of merging the two channels into one channel. One additional requirement is that the merged message format should be transparent to the destination processes, and be correctly split. In our refinement, we use the process *Merge* to realize merge, and the process *Split* to realize split, as illustrated in figure 14. This may decrease the overhead of arbitrating resource sharing, for example, at RNI_1. In particular, it may benefit for synchronizing the two consumers Q_1 and Q_2. And one virtual circuit suffices.

6 A Tutorial Example

6.1 The Audio Amplifier

We use an audio amplifier as a tutorial example to illustrate our refinement steps. The amplifier regulates the audio input signal in response to the button levels. It is structurally decomposed into three functional blocks illustrated in figure 15. The audio sampling rate is 64K bps. There are two buttons "+" (Up) and "-" (Down) used to increase and decrease the amplification ratio, respectively. The maximum rate of button press is once per second.

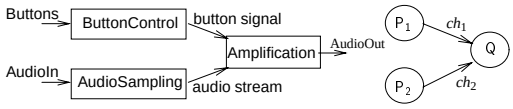


Figure 15: The audio amplifier and its process network

There are two channels ch_1 and ch_2. Both work as the producer-consumer pattern. The system requires acknowledgment for the audio channel ch_1. The button channel ch_2 does not need acknowledgment, but the button signals must be delivered in order (to keep causality) within tolerable period. In the system specification model, the audio output responds to the button press synchronously.

6.2 The Distributed Amplifiers

Analyzing the channel characteristics, we know that the two channels, ch_1 and ch_2, need to use a virtual circuit service. As we have refined the general producer-consumer model onto the virtual circuit services, we can choose one of the refined models for both channels. According to the system requirements, we adopt the producer-consumer model after channel refinement (figure 9) for the button channel ch_1, and the acknowledged producer-consumer with windowing (figure 12) for the audio channel ch_2.

Then we map the two refined producer-consumer models onto a NoC. Since the button channel uses less bandwidth, and the button signals contain less information bits, we assume that it is possible to converge and merge it with the audio channel. As a result, there are four choices for channel mapping:

(1) Map the producers P_1, P_2 and the consumer Q to three resources R_1, R_2 and R_3, respectively. Both channels, ch_1 and ch_2, have their own virtual circuit, vc_1 and vc_2, respectively.

(2) Map the producers P_1 and P_2 to the resource R_1, the consumer Q to another resource R_2. Both channels, ch_1 and ch_2, maintain their own virtual circuit, vc_1 and vc_2, respectively.

(3) Mapping is the same as in case (2). But the two channels share one virtual circuit.

(4) Mapping is the same as in case (2). But the two channels are merged into one channel.

We only take case (2) to show the whole refined system model in figure 16. Ideally the model should be plugged into the interface models offered by the network service layer, specifically the RNI's model and the switch's model. Different models yield different results. In our experiments, we choose a parallel-to-serial conversion process P/S if there is a need to arbitrate resource sharing, the points A and C. We choose a one-input two-output router process R to separate channel messages according to the channel id, the points B and D. The refined models for the other three cases can be derived accordingly.

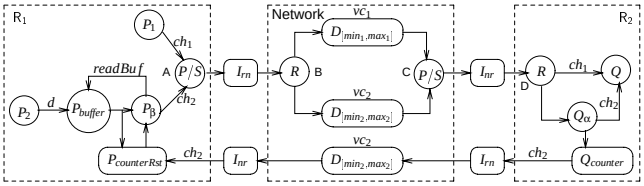


Figure 16: The refined amplifier model (Case (2))

Given the same set of parameters such as channel delays and window sizes, we have compared the four cases in terms of the average response delay of a button press to the amplification. In cases (1) and (2), the button signals may arrive ahead of the audio stream sampled during the buttons pressed, resulting in amplifying the previous sent audio data. This is because the two channel messages are delivered in different virtual circuits independently. In cases (3) and (4) the two channels can be synchronized since they share one virtual circuit. We can at least conclude that sharing virtual circuit may facilitate synchronization between channels. However, the virtual circuit bandwidth has to be high enough (case (3)), and the message format must be able to contain enough information (case (4)). Alternatively, data compression and de-compression may be introduced.

7 Conclusion

With the producer-consumer interaction pattern, this paper presents refinement procedures from perfectly synchronous communication onto NoC communication. During the refinements, application requirements such as reliability and throughput are satisfied. In ForSyDe, all the refinements are conducted within the functional domain, and are an integral design phase towards implementation.

The future work will focus on the refinements for the other three process interaction patterns defined in section 4.1, and consider the mixed effects in an application process network comprising two or more of the four process interaction patterns.

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 *Proc. of 2003 Conference on Embedded Software*.

[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*, 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):18, September 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] G. R. Andrews. Foundations of Multithreaded Parallel and Distributed Programming *Addison Wesley Longman, Inc.*, 2000.

[8] 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.

[9] P. Knudsen and J. Madsen. Integrating communication protocol selection with hardware/software codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1077 – 1095, 1999.

[10] 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*, March 2000.

[11] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Depretter. System level design with SPADE: an M-JPEG case study. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[12] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[13] I. Sander and A. Janstch. Transformation Based Communication and Clock Domain Refinement for System Design. In *Proc. of the 39th Design Automation Conference*, 20(1):281–286, June 2002.

[14] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17-32, Jan. 2004.

[15] S. Kumar et. al.. A Network on Chip Architecture and Design Methodology, In *IEEE Computer Society Annual Symposium on VLSI*, 2002.

[16] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.