Translating Communicating Sequential Processes to Formal System Design Models

Fangyi Shi

Kongens Lyngby 2011
IMM-MSC-2011-50

Summary

This thesis will focus on translating CSP processes to ForSyDe models, before translation, we will introduce some basic operators and typical processes in CSP for which we give translations to ForSyDe. As ForSyDe is a recent system-level model, we will also make a detailed introduction to it. After that, some example based translations will be illustrated to show how CSP processes to be translated to ForSyDe.

The CSP processes we considered in this thesis are primitive processes, processes with sending or receiving events, processes with choices (non-deterministic or deterministic choice), and processes containing a recursion. It involves parallelism problems, such as pairwise channel communication and deterministic choice control. Then the construction of CSP processes in ForSyDe will be generalized. Towards channel communication between two processes, we also discuss situations under Advanced Networks rather than Simple Networks.

We also propose an alternative approach to translate CSP processes to another low-level model, task graph. Compare with two approaches, we will find the advantage of ForSyDe.

All in all, we have achieved our goals but still have some problems left, which will be concluded in the end.

Preface

This thesis is prepared at Informatics Mathematical Modeling (IMM), the Technical University of Denmark (DTU) in fulfillment of the requirements for acquiring the MSC degree.

This thesis is initiated on February 9^{th} 2011 and finalized with the handover on August 9^{th} 2011, and is equivalent to a 30 ECTS credits course.

This thesis focuses on a method about translating a high-level model to a system-level model in theory, and it could be implemented on further research.

Fangyi Shi s090413
fangyis916@hotmail.com

Acknowledgements

I would like to thank my supervisor Michael Reichhardt Hansen, he invested amounts of time and effort in guiding me in the right directions and reading my work, he is a very respectable supervisor.

I also wish to thank Mikkel Koefoed Jakobsen, a Ph.D. student from IMM DTU. He helped me understand ForSyDe model deeply, and gave me a lot of useful materials.

Besides, I am very grateful to my parents, I cannot fulfill my study abroad without their supports.

Contents

Chapter 1

Introduction

1.1 Background

Nowadays, concurrent systems [1] become very common, thanks to the development of technology and our demands for computing power. A number of different activities can be carried out at the same time due to the concurrency. In computer science, many models can be used to describe and analyze concurrent systems, and one among them is called process calculus or process algebras [2]. Process algebras provide a tool for high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. Leading examples of process algebras include Communicating Sequential Process (CSP) [3], Calculus of Communicating Systems (CCS) [4], Algebra of Communicating Processes (ACP) [5]. In this document, we focus on CSP models to represent a high-level description for concurrent systems.

CSP is known as a well known concurrency model invented by Tony Hoare [6]. It helps to focus on the interactions of concurrent processes, such as the dining philosophers problem in [3]. One of big problems in a concurrent system is communication, so the way to describe communication issue becomes significant in CSP. We can describe communication channels, data types in channels, and data sending and receiving behaviors among concurrent processes in CSP.

However, data communication represented by CSP is modeled in high-level, we cannot know how these data communication is accomplished on a executive platform. It is very interesting to see how the high-level data communication performs on a low-level architecture. Therefore, a new low-level model called Formal System Design (ForSyDe) [7] is introduced. It is an abstract hardware model and based on processes and signals. Processes are connected by signals, each process in ForSyDe could be considered as an independent unit in a concurrent network.

Therefore, a translation from CSP to ForSyDe model is to show how CSP works on this particular parallel hardware architecture, and give a concept about the possibility of translating such kind of high abstract description to a system-level modeling framework.

1.2 Motivation

The motivation of this thesis comes from three aspects:

- Since ForSyDe is a new model for system design, there seems to be few articles to translate from other models to it. It is supposed to be a kind of challenge to translate CSP to this system-level modeling framework.
- Add an expression to ForSyDe framework. As we know that CSP description is widely used in many fields, if we can use ForSyDe framework to express CSP, the use of ForSyDe could be wider.
- CSP is related to a model in which all places with massive parallelism and this translation can relate conceptual parallelism with hardware parallelism.

1.3 Goals

Two main goals are supposed to be accomplished:

- Understand basic principles of ForSyDe model and discover difficulties it may occur when translating CSP to ForSyDe model.
- Select fragments of CSP for which there exist related ForSyDe frameworks.

Besides, there is a sub goal as well. ForSyDe is under an ongoing development and the document available is rather informal and insufficient, therefore, a sub goal is to make a careful introduction to ForSyDe, in order to help others understand ForSyDe framework better.

1.4 Structure of the thesis

In the rest of this document, it includes the following chapters.

- Chapter 2 gives a description of CSP, and a definition of CSP network. It will illustrate some subsets of CSP for which we give translations to ForSyDe.
- Chapter 3 makes a detailed introduction to ForSyDe, it shows how this kind of system-level framework works with only processes and signals.
- Chapter 4 gives us some simple typical examples to illustrate a main idea about translating CSP processes to ForSyDe models.
- Chapter 5 is to generalize how to construct ForSyDe models under a Simple CSP Network.
- Chapter 6 is to discover a way to translate an Advanced Network communication.
- Chapter 7 offers an alternative approach about translating from CSP to another low-level model, task graph [8]. After translation, the related task graph will be scheduled on different multi-core platforms.
- Finally, a conclusion of the whole project is in Chapter 8, it will conclude successes and declare difficulties during the project.

Chapter 2

Communicating Sequential Processes

In this chapter, we will make a brief introduction to CSP, including an overview of CSP, principle operators, several types of CSP processes which will be translated later, a definition of CSP network, and traces of CSP.

2.1 Overview

CSP is a formal language for describing patterns of interaction in concurrent systems. [1] Briefly speaking, CSP allows the description of systems in terms of component processes that operate independently, and interact with each other respectively through message-transmitting communication. Here, some concepts in CSP are listed as follows.

Events in CSP are used to describe a kind of behavior, and an event could include any action. For example, in a simple vending machine [3], two events can be defined.

- coin - the insertion of a coin in the slot

- choc - the extraction a chocolate from the dispenser of vending machine

However, there is no timing description of occurrences of events, thus, we don't know how long every event will last.

There is one kind of special event in CSP, which is called *channel event* in this thesis. From the name, we can see that it is an event contains a channel with sending or receiving action.

Alphabet is considered as a set of names of events which are relevant for a particular description of an *object*. Objects are realistic objects in the world around us, such as a simple vending machine is an object, and its alphabet here is the set {coin, choc}.

Processes in CSP are to stand for the behavior pattern of an object, and it can be described in terms of the limited set of events which are selected in its own alphabet. [3]

A special process which represents fundamental behaviors is called *primitive processes*. Two of instances of primitive processes are STOP (the process that communicates nothing) and SKIP (represents successful termination).

Events and primitive processes are two kinds of basic components in a process algebra.

2.2 Operators

Operators which are going to be translated to ForSyDe models will be enumerated below.

- Prefix
  Let a be an event and P be a process, $a \rightarrow P$ is to represent a is a prefix of P. After finishing event a, process P will start.
- Sending
  If c is a channel, and v is a valid data through c ($v \in \alpha(c)$). $c!v$ is a notation to represent sending v via c.
- Receiving
  If c is a channel, and x is a valid data through c ($x \in \alpha(c)$). $c?x$ is a

notation to represent receiving a valid data via c.

- Non-deterministic choice
  The non-deterministic (or internal) choice operator is used to define a process which exhibits a range of possible behaviors, and the internal mechanism choose which one to perform. A non-deterministic choice between processes P and Q is with the notation $P \sqcap Q$. Choose P or Q is nothing related to environment control.

- Deterministic choice
  The deterministic (or external) choice operator is also to define a process which exhibits a range of possible behaviors, but the environment controls the choice. Notation $P \Box Q$ is to represent deterministic choice between process P and Q. Take $P \Box Q$ as an example, if environment only allows P take place, it will always choose P, however, if it is non-deterministic choice $P \sqcap Q$ instead, it may select Q to execute by accident, and then cause deadlock.

  Note: we also use a notation (|) to represent a choice if there is no specific definition about internal or external. In this thesis, we will clarify a choice operator either internal choice or external choice.

- Recursion
  If a behavior of a process is endless and repetitive, it is tedious to write down the entire behavior of process, so we need a method to describe such recursion. If a process P always repeat event a, P could be defined as $P = a \rightarrow P$. As we can see, the method is to use a prefix notation ($\rightarrow$) and point to original process again. The recursion we discussed in this thesis is called *guarded recursion*, which means the definition will work only if the right-hand side of the equation starts with at least one event prefixed to all recursive occurrences of the process name. Therefore, the equation $P = P$ does not succeed in defining anything. [3]

- Concurrency
  CSP is introduced to describe concurrent system, and the notation to describe concurrency between processes P and Q is $P \parallel Q$. When P and Q are assembled to run concurrently, events that are in both alphabets require simultaneous participation of both P and Q. However, events which are only in the alphabet of P are of no concern to Q, and these events can occur independently of Q whenever P engages in them. Similarly, Q can engage alone in events only in alphabet of Q. [3]

However, there are many other operators in CSP we will not consider their ForSyDe models in this thesis, for example:

- The hiding operator ($\setminus$), a process $(a \rightarrow P) \setminus \{a\}$ assumes that event a doesn't appear in process P, it could be simply reduced to P.

- The interleaving operator ($|||$), so for the process $P ||| Q$, it behaves as both P and Q simultaneously. The events from both processes are arbitrarily interleaved in time.

- The chaining operator ($\gg$), and $P \gg Q$ represents P and Q are joined together by an internal channel, so that the sequence of messages output by P and input by Q on this internal channel is concealed from their common environment. [3]

2.3 Processes in CSP

A CSP process is a sequential process which is consist of events and primitive processes. Sometimes, an equation to define a process may contain other processes or itself. However, any process can be represented by events and primitive processes basically. For instance,

$P = a!5 \rightarrow Q$
$Q = b?x \rightarrow SKIP$

We can find process Q is consist of a channel event and a primitive process, and process P contains a channel event and process Q. Process P also can be defined as:
$P = a!5 \rightarrow b?x \rightarrow SKIP$

As a result, P is made up of two channel events and a primitive process basically.

During translation, we focus on two types of sequential processes.

- Binary selective process: a process with the structure $X \Box Y$ or $X \sqcap Y$, where X and Y are two sequential processes. For example, a binary selective process P:

  $P = (a?x \rightarrow STOP) \Box (b?y \rightarrow STOP)$

- Recursive process: a process contains a recursion, and an example of a recursive process P is below:

  $P = c!3 \rightarrow d?x \rightarrow P.$

2.4 CSP network

2.4.1 Definition

CSP network is to make parallel CSP processes together in one network. Below is a CSP network, CSP processes $P_1, P_2, P_3, ..., P_n$ are in parallel with each other.

$Network = P_1 \parallel P_2 \parallel P_3 \parallel ... \parallel P_n$

Where, P_i (i∈[1,n]) can include any non- parallel CSP notations, such as sending notation (!), prefix ($\rightarrow$), internal choice ($\sqcap$) etc.

Each P_i has its own input alphabet $in(P_i)$ and output alphabet $out(P_i)$. $in(P_i)$ is a set of channels on which P_i can receive data, and $out(P_i)$ is a set channels on which P_i can send data. $in(P_i)$ and $out(P_i)$ could be empty ($\emptyset$), if no input channel or output channel is involved in P_i. For example, a CSP description of process P_1 is below:

$P_1 = c?x \rightarrow d!2 \rightarrow e?y \rightarrow f!5 \rightarrow STOP$

We can find that channels c and e are channels waiting for data, and channels d and f are sending data. So $in(P_1) = \{c, e\}$; $out(P_1) = \{d, f\}$.

Some limitations on $in(P_i)$ and $out(P_i)$ of any P_i in a CSP network $N = P_1 \parallel P_2 \parallel ... \parallel P_n$ $(i \in [1, n], n \geq 1)$ are as follows:

- The input channel set and output channel set cannot be overlapped, thus, $in(P_i) \cap out(P_i) = \emptyset$. For example, $P_i = (c?x \rightarrow STOP) \Box (c!8 \rightarrow STOP)$ is not allowed.

- If a channel $c \in in(P_i)$, c cannot occur on the input channel set of any other process $P_j, j \neq i$, thus, $in(P_i) \cap in(P_j) = \emptyset$. It has the same limitations on output channel set, $out(P_i) \cap out(P_j) = \emptyset$. For instance, if P_2 and P_3 in Network N are defined:
  $P_2 = c?x \rightarrow P_2$
  $P_3 = c?y \rightarrow STOP$

  We can find $in(P_2) \cap in(P_3) = \{c\}$, so N is not discussed in this thesis.

According to the frequency of occurrence of one channel in a process of CSP network, there are two kinds of CSP network in this thesis:

- *Simple Network*, for any channel $c \in in(P_i) \cup out(P_i)$, if only one occurrance of channel event through c in P_i, such kind of network is called Simple Network.

- *Advanced Network*, it does not have such kind of constraint on channels as Simple Network. It allows channel events through channel c ($c \in in(P_i) \cup out(P_i)$), occurring several times in P_i.

2.4.2 Simple Network example

A simple vending machine [3] is a typical example for Simple Network, which is to serve a cup of chocolate after inserting a coin. Its network (VM) involves two processes, process *order* is defined from customer's side and the other process *makechoc* is defined from the vending machine's side. Two events *coin* and *choc* appeared in VM, are described in Sect. 2.1, besides, one channel *ch* is used to transmit data $\{start\}$ between the two processes.

The simple CSP network $VM = order \parallel makechoc$

$order = coin \rightarrow ch!\{start\} \rightarrow order$
$makechoc = ch?\{start\} \rightarrow choc \rightarrow makechoc$

We can find that:
$in(order) = \emptyset$, $out(order) = \{ch\}$; $in(makechoc) = \{ch\}$, $out(order) = \emptyset$

Channel *ch* only appears once separately at processes *order* and *makechoc*, so VM is a simple CSP network.

2.4.3 Advanced Network example

An Advanced Network example (ADNet) is as follows:

$ADNet = P_1 \parallel P_2$
$P_1 = c!4 \rightarrow c!3 \rightarrow P_1$
$P_2 = c?x \rightarrow c?y \rightarrow c?z \rightarrow P_2$

We can find sending events through channel c appear twice in P_1, and receiving events through channel c appear three times in P_2, so $ADNet$ is an Advanced Network.

2.5 Trace

A trace of the behavior of a process is a finite sequence of sequence of symbols recording the events in which the process has engaged up to some moment. A trace is denoted as a sequence of symbols, separated by commas and enclosed in angular brackets. [3] Notice that <> is the empty sequence containing no events.

For example, a CSP process P is defined:

$P = a \rightarrow b \rightarrow P$

Since there is a circle in process P, the sets of possible trace are infinite, and up to some certain moments it could be:
<>, before process P has engaged in any events.
$< a >$, before executing the first event b.
$< a, b >$, finish executing the first circle.
$< a, b, a >$, before the event b in the second circle.
$< a, b, a, b >$, finishing executing the second circle.
...

For channel events, if it is a sending event such as $c!v$, the related symbol in trace is denoted as $c{:}v$. If it is a receiving event as $c?x, x \in Z$, the related symbol in trace is denoted as $c{:}x,\ x \in Z$. When $c!v$ and $c?x$ communicates with each other, only v is transmitted. As a result, only event $c{:}v$ will appear in trace after concurrency.

Take the CSP network *VM* in Sect. 2.4.2 for example, the possible sets of trace up to some certain time are as follows:

<>, before process VM has engaged in any events.
$< coin >$, before communicating through channel ch for the first customer.
$< coin, ch : \{start\} >$, before serving the first cup of chocolate.
$< coin, ch : \{start\}, choc >$, finish extracting the first cup of chocolate.
$< coin, ch : \{start\}, choc, coin >$ before communicating through channel ch for the second customer.
...

During translation, every trace of the CSP process can be reconstructed from the signals in the corresponding ForSyDe model. We will give an example in Sect. 4.8. However, we will not prove the correctness of the construction in this thesis.

Chapter 3

Formal System Design

In this chapter, we will give an introduction to ForSyDe models. It includes a brief overview, signals and three sorts of processes in ForSyDe framework. Many simple examples will be used to help to express ForSyDe models.

3.1 Overview

ForSyDe is an abbreviation of Formal System Design, which is a methodology aimed at raising the abstraction level in system-level design, e.g. System on Chip Systems, Hardware or Software. The components of ForSyDe systems are *processes* and *signals*. In short, it is a system which is modeled as a network of processes interconnected by signals. Figure 3.1 illustrates an example of a ForSyDe framework, which contains two processes (P, Q) and three signals (S1, S2, S3). This kind of ForSyDe framework with individual processes connected by directed signals is called *process network*. It is a kind of data flow paradigm, under which algorithms are described as directed graphs where the blocks represent computations (or functions) and the arcs represent data paths. [9]

A signal could contain a sequence of elements called *tokens*, and in ForSyDe it must indicate how many tokens are taken from an *input signal* and how many

tokens are sent through an *output signal*. It is indicated by the number at the head and end points of a signal. Take Figure 3.1 for example, it shows P takes one token from signal S1, and outputs one token through S2.

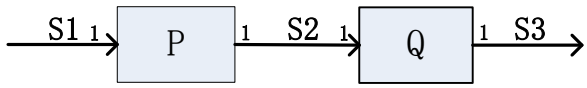


Figure 3.1: an example of ForSyDe framework

A concrete example of Figure 3.1 is given below.

Example 1
S1 contains five positive numbers $< 1, 2, 3, 4, 5 >$, since an element in a signal is called a token, S1 has five tokens. As an input of P, one token is taken at a time in order. Process P is a process which is defined by a function $f(x) = x^2$, and Q is also a process which is defined by a function $g(y) = y + 1$.

A behavior of the process network is given by a set of signals:

S1:< 1, 2, 3, 4, 5>
S2:< 1, 4, 9, 16, 25>
S3:< 2, 5, 10, 17, 26>

Note: the sequence of tokens in a group indicates that these tokens take place one after another, and time interval between two tokens is one time unit. In Example 1, P gets an integer *1* from S1 at the beginning, and after one time unit P takes *2*.

If a minor modification is done to S2 in Figure 3.1. Change the integer at the end of the signal S2 from *1* to *2*, shown in Figure 3.2. It means process Q will take two tokens at a time. Because of the modification, S3 will get one token when S2 has two tokens. and the speed of producing tokens of S2 is twice faster than S3. If only one token left, Q will not receive any. As a result, the function inside Q will contain two variables, such as $g(x, y)$. A concrete example is given below on the model in Figure 3.2.

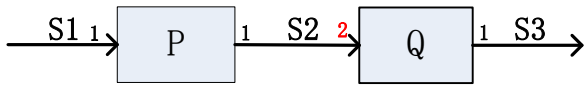


Figure 3.2: an example with a modification to S2 in Fig. 3.1

Example 2
Initialize input signal S1: <1, 2, 3, 4, 5>.
The functions in P and Q are:
$P : f(x) = x^2$
$Q : g(x, y) = y - x$

As a result, the behavior of the process network is:

S1:<	1,	2,	3,	4,	5>
S2:<	1,	4,	9,	16,	25>
S3:<		3,		5>	

It is easy to find that tokens in S3 have a wider space than S1 and S2, because tokens which are in the same column take place at the same time. The result shows that at the first time unit, only S1 and S2 carry tokens, then at the second time unit, S3 has a token with the value *3*, as well as S1 has a token *2* and S2 has a token *4*. We can find that it still has a token with value *25* left in S2, but Q will not accept it due to Q needs to take two tokens at a time.

3.2 Signals

From examples above, we can see that a *signal* can be defined as a sequence of tokens, with a given type. It is classified as *input signal*, *output signal* and *internal signal* in a process network. In Figure 3.1, S1 is an input signal, S2 is an internal signal connecting P with Q, and S3 is an output signal.

Graphically, a signal is represented as an arrowed line. According to the direction of arrow, it is easy to detect the start and end point of a signal.

Generally, the sequence of tokens could be infinite or finite. A token is a value of a given type. The type could be any algebra type of a programming language. It could be integer, string, or any compression type, but one signal cannot carry values belonging to different types. In Example 1 and 2, the type of tokens is integer, and five tokens in S1 are 1, 2, 3, 4 and 5.

Two special types of tokens are involved in this thesis:

- tokens for triggering, recorded as s, and they are only used to trigger next processes without any meanings of data.
- tokens for terminating, recorded as low case t, they are only used to represent output tokens of process $STOP$. Since primitive CSP process $STOP$

is a terminated CSP process, this type of tokens is a symbol for terminating the entire procedure.

In ForSyDe, it is important to notice that there is no time consumption in signal channel. Therefore, in Figure 1, whenever S2 gets a token from P, this token will be transferred to input point of Q concurrently.

3.3 Processes in ForSyDe

The definition of Processes in [7] is: "Processes are pure functions on signals, i.e. for a given set of input signals a process always gets the same set of output signals. It can also be viewed as a black box which performs computations over its input signals and forward the results to adjacent processes through output signals."

A process with a function is called *pure process*. A pure process can only have one output, for instance, processes P and Q in Example 1 and 2 are pure processes. However, the process network in Figure 3.1 could also be considered as one process, a *hierarchical process*. Therefore, a hierarchical process could contain internal states with one or more processes. We call hierarchical process as process for short in the rest of this document for convenience. Some typical styles of processes are discussed below.

3.3.1 General Process

A general process is supposed to take one or more signals as its inputs and outputs. Figure 3.3 illustrates a general process with n input signals and m output

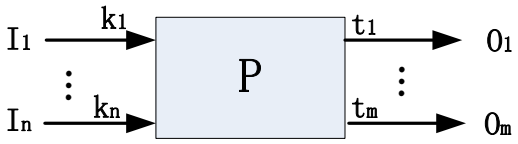


Figure 3.3: a general process model

signals. An input signal (I_j) need to indicate the number of tokens (k_j) that a process P will take from it, where $j \in [1, n]$, $n > 0$. Similarly, t_i is the number of tokens through output signal O_i, where $i \in [1, m]$, $m > 0$. A concrete example

is below with a model in Figure 3.4.

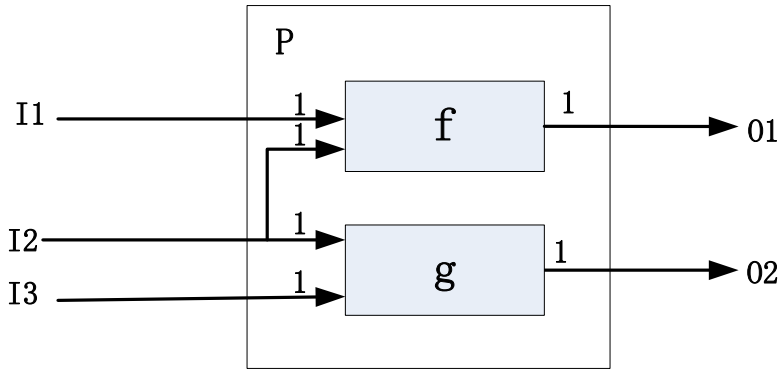


Figure 3.4: an example of general process

Example 3
P is a process with two functions:
$f(v_1, v_2) = v_1 + v_2$;
$g(v_2, v_3) = v_2 - v_3$;
where v_j represents the related token of I_j.

O1: output value of $f(v_1, v_2)$;
O2: output value of $g(v_2, v_3)$.

Now, specify inputs,
I1:< 1, 3, 5, 7>
I2:< 1, 2, 3, 4>
I3:< 2, 2, 1, 1>

After calculation in P, outputs are:
O1:< 2, 5, 8, 11>
O2:< -1, 0, 2, 3>

There are two common properties of general processes:

- Functions in general processes are supposed to take no time delay, in other words, once it can accept tokens from input signals, it will produce output signals immediately.
- All of inputs of one pure process should be synchronized. In Figure 3.4, the only way to execute process g is that I2 and I3 are both ready to send

a token to it, otherwise it will wait. For instance, if we change the input signals I3 in Example 3 with only three tokens. Now, the new example is illustrated below in Example 4.

Example 4
Initialize three input signals as:
I1:< 1, 3, 5, 7>
I2:< 1, 2, 3, 4>
I3:< 2, 2, 1>

The behavior of the outputs is:
O1:< 2, 5, 8, 11>
O2:< -1, 0, 2>

Although, I2 has already prepared for the fourth value, I3 doesn't contain one more value at the moment. As a result, process *g* will not accept values from any of input signals until I3 is ready for one more token.

3.3.2 Delay Process

As we mentioned before, a general process does not cost any time consumption, but a *Delay Process* is a kind of process that consumes time. A single Delay Process is in Figure 3.5, with one input signal *In* and one output signal *Out*.

Figure 3.5: A ForSyDe model for a Delay Process

For a Delay Process, it contains an initial token in output signal, so the tokens come from its input signal will always be one time unit delayed. A sample of signals is below:

Example 5
In: < s1, s2, s3, s4>
Out:< s0, s1, s2, s3, s4>

s0 in signal *Out* is the initial token of the Delay Process, we can find that at the first time unit, the Delay Process will output *s0*, then forward the tokens from signal *In*.

A common usage of a Delay Process in ForSyDe is use it to control a loop, an example which contains a loop could be modeled in Figure 3.6. Process *P* needs

the output value to be another input value of itself, and the Delay Process is used to send an initial token $< P_0 >$ to trigger the entire procedure. If we remove the Delay Process in Figure 3.6, the output signal will connect with input signal directly, shown in Figure 3.7.

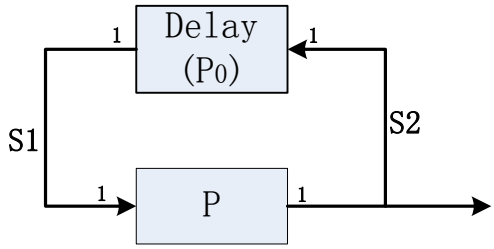


Figure 3.6: an example with a loop

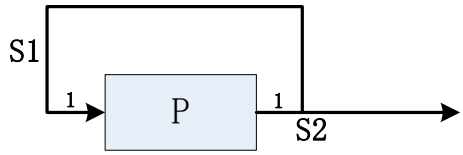


Figure 3.7: an example contains a loop without a Delay Process

However, the model in Figure 3.7 is illegal in ForSyDe, here, we illustrate an example below.

Example 6
$P : f(x) = x + 1;$

We assume process P can send an initial token $< 0 >$ through S2 to trigger the entire procedure, and the token $< 0 >$ will go back to input point of P at the first time unit. So without any time consumption, P takes a token $< 0 >$ from S1, sends $< 1 >$ through S2, and then P also has to get the new input $< 1 >$. All these procedures are supposed to be fulfilled at the first time unit.

The behavior of signals is:
S1:$< 0(1) >$
S2:$< 0(1) >$

At the first time unit, P is supposed to take two tokens $< 0 >$ and $< 1 >$, but the index at the end of signal S1 is 1, then P is not allowed to take two tokens at the same time, therefore, a Delay Process is necessary.

Example 7 below is with the model in Figure 3.6.

Example 7
$P_0 = 0$;
Delay: it consumes one time unit, and replicates to forward values from S2 to S1 100 times.
P is defined by a function $f(x) = x + 1$.
S2: output value of $f(x)$.

The behavior of signals is:
S1:< 0, 1, 2, 3, ..., 100>
S2:< 1, 2, 3, 4, ..., 101>

As a result, in ForSyDe model there is a requirement that every circle contains a Delay Process, in this way to avoid inconsistent problems like Example 6 above. Notice that a Delay Process always costs one time unit, but we could combine several Delay Processes together to consume one or more time units.

3.3.3 Choice Process

In ForSyDe, there is a pair of choice operators, shown in Figure 3.8 related to Choice1 and Choice2, and they are connected by a signal T. According to signal T, both Choice1 and Choice2 will agree on the upper signals or the lower signals to use, and for sure both choice operators will get the same signal from T. For example, signal T can provide a Boolean value, true or false, if T sends true, Choice1 will choose to send tokens through *up1* and Choice2 will choose to receive tokens through *up2*. On the contrary, if T sends false, Choice1 will pick up *down1* to send tokens and Choice2 will select *down2* to get tokens. Process1 and Process2 will not affect the selection. Only the input signal T controls the whole selection.

Constraints on index $(k_1, k_2, ..., k_{10})$ beside each signal are defined below.

- k_1: an arbitrarily integer $x \in [1, \infty)$
- k_2: either the same value as k_1 or zero
- k_3: either the same value as k_1 or zero
- k_4: the same value as k_1
- k_5: the same value as k_1

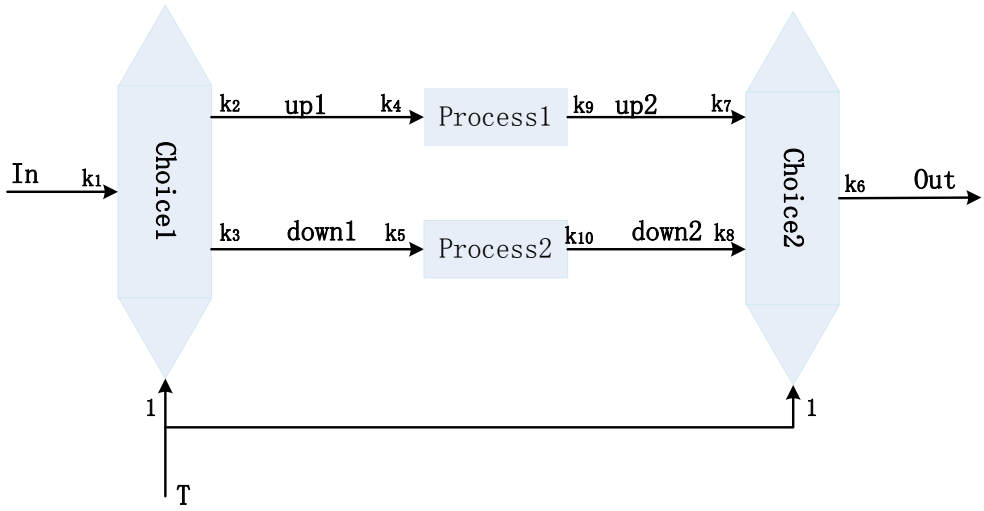


Figure 3.8: a Choice Process model

- k_6: an arbitrarily integer $y \in [1, \infty)$
- k_7: either the same value as k_6 or zero
- k_8: either the same value as k_6 or zero
- k_9: the same value as k_6
- k_{10}: the same value as k_6

It is important that k_2 and k_3 cannot keep the same value at the same time, which means if k_2 is the same as k_1, k_3 must be zero, and it is the same on the contrary. So as k_7 and k_8, they cannot keep the same value at the same time too. It is because when we decide upper or lower signals to use by input signal T, we need the selected signals to transmit tokens, and the non-selected signals do not get any tokens. If we choose up1 and up2, $k_2 = k_1, k_3 = 0$ and $k_7 = k_6, k_8 = 0$, otherwise, when down1 and down2 are selected, $k_3 = k_1, k_2 = 0$ and $k_8 = k_6, k_7 = 0$.

A concrete example is illustrated in Example 8 with the model in Figure 3.9.

Example 8
T: Sends a Boolean value randomly, token $< T >$ represents true and $< F >$ represents false.
Choice1: If T sends true, up1 is selected, so Choice1 gets ready to forward a

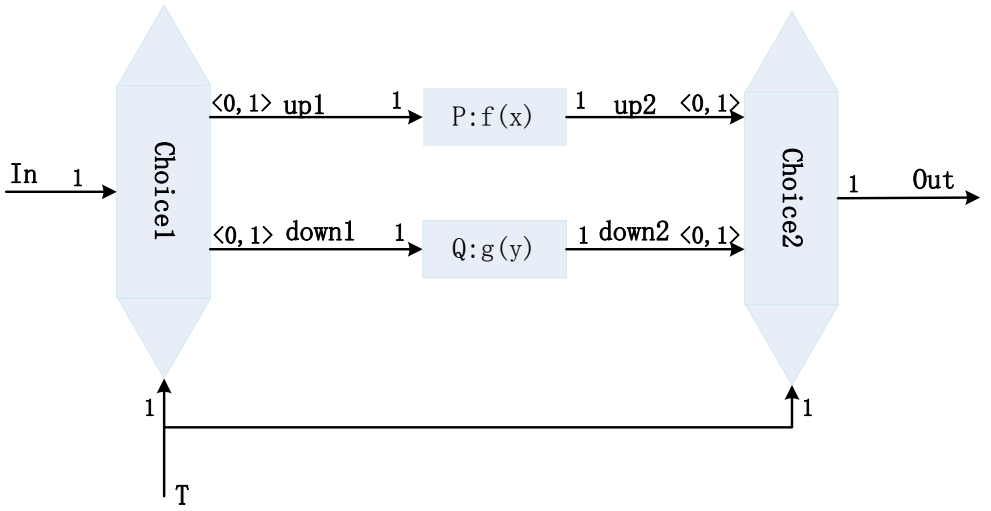


Figure 3.9: an example of a Choice Process

token from signal In to up1. Otherwise, down1 is selected to get ready instead. Choice2: If T sends true up2 is selected, so Choice2 gets ready to forward a token from up2 to signal Out. Otherwise, down2 is selected to get ready instead.

Functions in process P and Q are defined as:
$P : f(x) = x^2$
$Q : g(y) = y + 1$

Initialize the input signals as:

In:<	1,	2,	3,	4,	5>
T :<	T,	F,	F,	T,	F>

The behavior of signals is performed as follows:

In:<	1,	2,	3,	4,	5>
T :<	T,	F,	F,	T,	F>
up1:<	1,			4>	
down1:<		2,	3,		5>
up2:<	1,			16>	
down2:<		3,	4,		6>
Out:<	1,	3,	4,	16,	6>

Chapter 4

Example based translations from CSP to ForSyDe

In this chapter we will give examples to show how CSP processes to be translated to ForSyDe frameworks. Primitive process and several non-parallel operators in CSP will be described in ForSyDe model separately. In the next chapter the translation will be generalized.

4.1 Primitive processes

For a single primitive process, the way to translate it is to map such kind of CSP processes to pure processes in ForSyDe with one input signal and one output signal. For example, primitive CSP processes $STOP$ and $SKIP$ are modeled in Figure 4.1.

The input signals and output signals are:

- S_{start_stop} and S_{start_skip}: a start signal to declare that this process is ready to proceed.

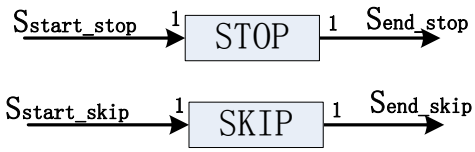


Figure 4.1: ForSyDe models for primitive processes $STOP$ and $SKIP$

- S_{end_skip}: a procedural output signal which is connected to next process or the environment.
- S_{end_stop}: this signal is never connected with other processes, since process STOP represents communicating nothing. It can just be observed by the environment.

4.2 Sending

A sending event $c!v$ which represents sending a value v from channel c can be modeled in Figure 4.2.

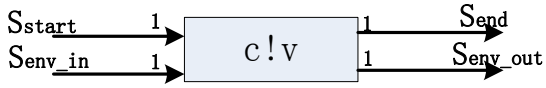


Figure 4.2: a sending event $c!v$ in ForSyDe

Two input signals and two output signals are involved in ForSyDe model:

- S_{start} : a start signal to declare that this process of sending event is ready to proceed.
- $Senv\_in$: an environment control input signal coming from its concurrent partner to control the synchronization. The concurrent partner for sending part is a receiving part of the same channel.
- $Senv\_out$: an environment control output signal to control other processes which are dependent on this process. This signal is also connected to a receiving part of the same channel.
- S_{end} : a procedural output signal which is connected to next process or the environment.

4.3 Receiving

As a receiving part of a communication channel, it has the same model structure as a sending part, which includes two input signals and two output signals. Take $c?x$ for example, it is modeled in Figure 4.3, where x could be any valid transmitting through channel c.

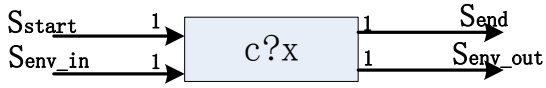


Figure 4.3: a receiving event $c?x$ in ForSyDe

Four signals play similar roles as related signals in a process of sending event:

- S_{start} : a start signal to declare that this process of receiving event is ready to proceed.
- $Senv\_in$: an environment control input signal coming from its concurrent partner to transmit data. The concurrent partner for receiving part is a sending part of the same channel.
- $Senv\_out$: an environment control output signal to control other processes which are dependent on this process. Here, this signal is connected to a sending part of the same channel.
- S_{end} : a procedural output signal which is connected to next process or the environment.

4.4 Prefix

In ForSyDe, the way to represent Prefix symbol ($\rightarrow$) is to connect a procedural output signal of one process with a start signal of the next process.

For instance, a CSP description is that $P = c?x \rightarrow STOP$, where event $c?x$ is a prefix of $STOP$. Both ForSyDe models for event $c?x$ and primitive process $STOP$ have already discussed by Sect. 4.3 and 4.1. So a ForSyDe model for CSP process P is illustrated in Figure 4.4. In this example, the signal S_{c_end} starting from process $c?x$ and connecting with process $STOP$ will be related to the representation of a prefix symbol in CSP.

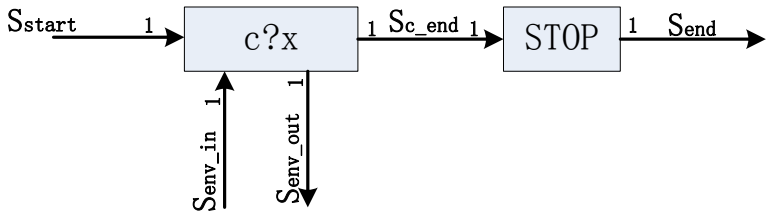


Figure 4.4: A ForSyDe model for $c?x \rightarrow STOP$

The definitions of signals S_{start}, S_{end_in} and S_{end_out} in Figure 4.4 are the same as S_{start}, S_{end_in} and S_{end_out} in Figure 4.3. S_{c_end} is an internal signal representing the end of process $c?x$ and a start signal of $STOP$. S_{end} is the same as S_{end} in Figure 4.1.

4.5 Non-deterministic Choice

Non-deterministic choice is a choice decided by internal system. There is a problem while translating, since ForSyDe models are deterministic models. Therefore, an extra simulator outside ForSyDe framework could be introduced to provide a non-deterministic mechanism. A concrete example is given below.

CSP description:
$P = (a?x \rightarrow STOP) \sqcap (b?y \rightarrow STOP)$

From CSP description, there is an internal choice in P between channel a and b. Once executing P, it will choose either channel a or channel b to be ready to receive data.

In ForSyDe, a pair of choice operators which we have discussed in Sect. 3.3.3 is used to represent a CSP choice. As a result, the related ForSyDe model for above example is shown in Figure 4.5.

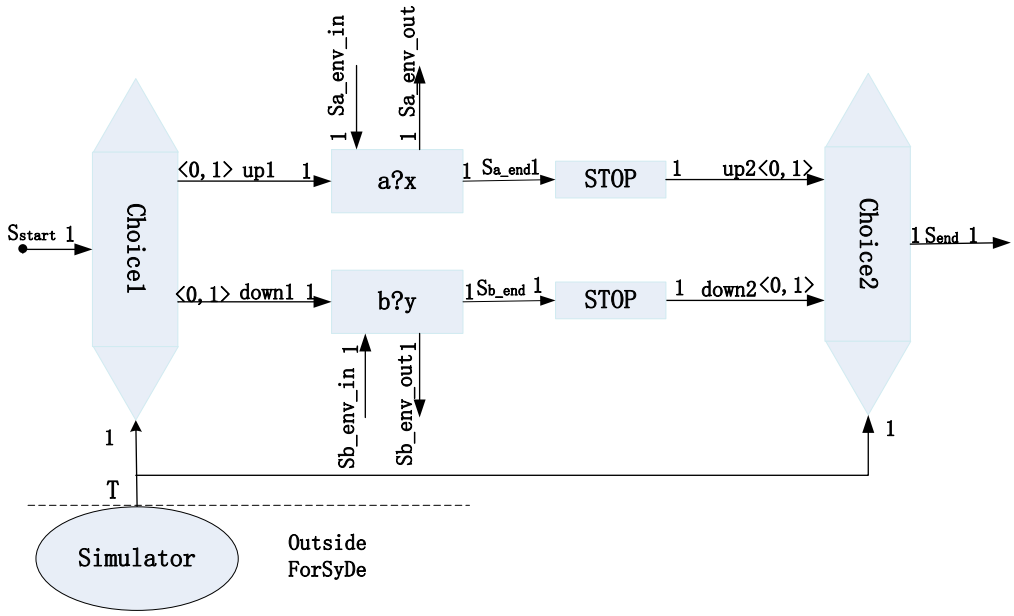


Figure 4.5: A ForSyDe model for $a?x \rightarrow STOP \sqcap b?y \rightarrow STOP$

Signals between Choice1 and Choice2 have been described in details in Sect. 4.4, and other signals are defined as follows:

- S_{start} : a start signal to declare that this choice process is ready to proceed.
- T : a control signal to choose whether $(up1, up2)$ or $(down1, down2)$ to be prepared for using. An extra simulator outside ForSyDe model will produce random tokens for the signal T to control the selection.
- S_{end} : a procedural output signal which is observed by the environment, since it forwards tokens from $STOP$ process which cannot be connected to other ForSyDe processes.

4.6 Deterministic Choice

The example for deterministic choice is similar to non-deterministic, only one operator's modification, so the CSP description is as follows:

$P = (a?x \rightarrow STOP) \,\square\, (b?y \rightarrow STOP)$

The ForSyDe model for deterministic choice is very similar to the non-deterministic choice model in Figure 4.5. The only difference between them is who is going to connect with signal T. In deterministic choice ForSyDe model, signal T may depend on another process or the environment, as we will see when we consider about parallelism in Sect. 4.9.

4.7 Recursion

A recursive process in CSP is translated to a ForSyDe model containing a circle. The simple example below shows how the translation of recursive process works. The performance of P will repeat to receive data through channel c.

CSP description:
$P = c?x \rightarrow P$

The event $c?x$ has already modeled in Figure 4.3, and the Delay Process is often added at the end of the circle. Thus, process P could be modeled in Figure 4.6.

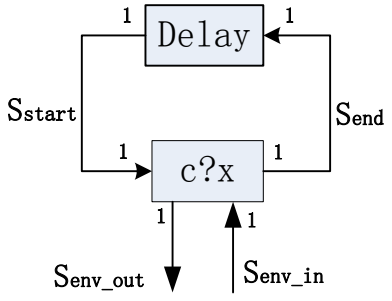


Figure 4.6: A ForSyDe model for $P = c?x \rightarrow P$

According to the property of Delay Process, S_{start} will contain one token at the beginning, so when S_{env_in} sends a token to the ForSyDe process $c?x$, $c?x$ will

be executed immediately. After one time unit delay, process $c?x$ will wait for another token from signal S_{env_in}.

4.8 Pairwise communicating channel

In a Simple Network $P_1 \parallel P_2 \parallel P_3 \parallel ... \parallel P_n$, which involves n processes in parallel. If there is a common channel in two different processes, one is for sending data and the other is for receiving data, then the related processes in ForSyDe model need to be connected with each other by their environment control signals to guarantee the concurrent communication.

A concrete example with CSP processes P_1 and P_2 is given to show how to construct a pairwise communicating channel.

CSP description:
$P_1 = c?x \rightarrow STOP$
$P_2 = c!2 \rightarrow STOP$
where c has integer type, i.e. $x \in Z$

P_2 is to send an integer *2* through channel c and then stop, while P_1 is to receive a value from channel c and stop.

As a result,
The set of traces of P_1 is $\{<>\} \cup \{< c : v > | \ v \in Z\}$;
The set of traces of P_2 is $\{<>\} \cup \{< c : 2 >\}$;
The set of traces of $P_1 \parallel P_2$ is $\{<>\} \cup \{< c : 2 >\}$

The idea behind the translation of a parallel CSP process is to translate the sending and receiving parts separately as we have seen examples of in Sect. 4.2 and Sect. 4.3, and then combine the two parts by connecting corresponding environment control signals. This will be shown below.

As we have already discussed before, CSP process P_1 and P_2 can be modeled in Figure 4.7 and Figure 4.8.

We can find there are two environment control signals S_v and S_s in process $c?x$, and so as in process $c!v$.

S_v: an environment control signal for communication from the sending part to receiving part.

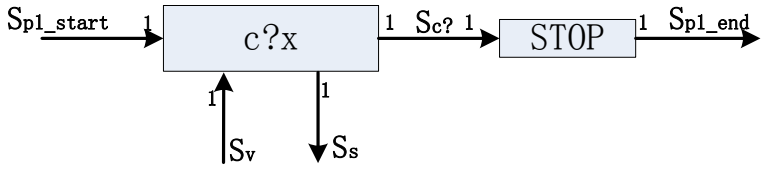


Figure 4.7: A ForSyDe model for $P_1 = c?x \rightarrow STOP$

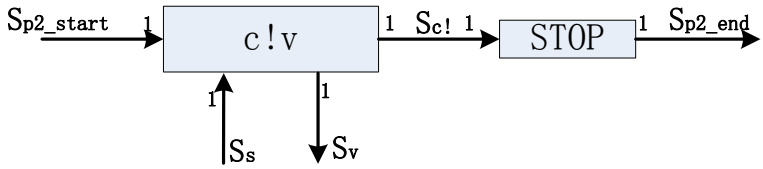


Figure 4.8: A ForSyDe model for $P_2 = c!2 \rightarrow STOP$

S_s: an environment control for synchronization from receiving part to sending part.

Therefore, in a parallel CSP network, two environment signals of the same channel are needed to be combined together. Here, combine S_v and S_s of P_1 and P_2, we can get a ForSyDe model for $P_1 \parallel P_2$ in Figure 4.9. In order to understand the mechanism of channel communication, we visualize the internal processes of the hierarchical process $c!v$.

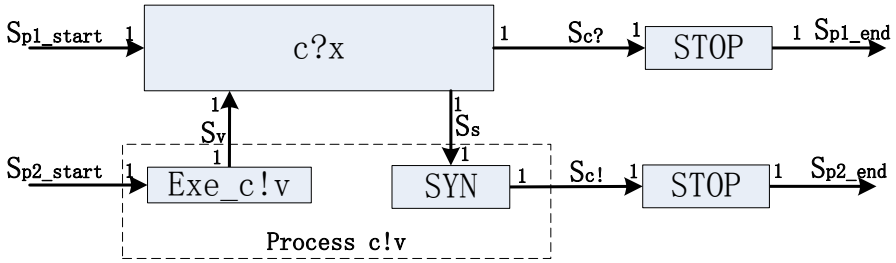


Figure 4.9: A ForSyDe model for $P_1 \parallel P_2$

Processes and signals in the ForSyDe model for $P_1 \parallel P_2$ are described below.

Processes

- $c?x$: a process for receiving a value from channel c, function inside is

$c?x(x, v) = v$, x and v are tokens from S_{p1_start} and S_v.

- $Exe\_c!v$: an internal process of $c!v$, it is used to sending tokens through channel c, function inside is $Exe\_c!v(x) = v$. When getting a token x from S_{p2_start}, output a value v, in this example, $v = 2$.
- SYN: an internal process of $c!v$, it is used to wait for a signal from the receiving part in this way to guarantee synchronization. Function inside is $SYN(v) = v$, which is used to forward a value from its input signal.
- $STOP$: a process for the primitive CSP process STOP.

Signals

- S_{p1_start}: a start signal of process $c?x$, when it contains a token, the process $c?x$ is ready to start.
- S_{p2_start}: a start signal of process $Exe\_c!v$, when it contains a token, the process $Exe\_c!v$ will start.
- S_v: an environment control signal starting from $Exe\_c!v$ and connecting with $c?x$, the tokens of S_v should be valid values for channel c.
- S_s: an environment control signal for synchronization from process $c?x$ to SYN.
- $S_{c?}$: an output signal from $c?x$ to the next process.
- $S_{c!}$: an output signal from $c!v$ to the next process.
- S_{p1_end}: an output signal of $STOP$ in P_1.
- S_{p2_end}: an output signal of $STOP$ in P_2.

Note: tokens in S_{p1_start} and S_{p2_start} are tokens for triggering, and tokens in S_{p1_end} and S_{p2_end} are tokens for terminating. Besides, tokens in the other signals here are all integer type.

The trace of the parallel process can be observed on $S_{c?}$ and actually also on $S_{c!}$, as values will appear on these signals when the sending and receiving parts are synchronized.

The procedure of P_2 is as follows, once there is a signal from S_{p2_start}, process $Exe\_c!v$ will be executed, $Exe\_c!v(x) = 2$, and send an integer 2 to process $c?x$ via S_v. Then it will wait a token from $c?x$ to start the SYN, $SYN(v) = v$. So

a token will be sent through $S_{c!}$ to $STOP$, finally the terminal token will be sent via S_{p2_end}.

While the procedure of P_1 is that, when there is a data from S_v, another signal S_{p1_start} must also has a token at that moment so as to start executing $c?x(x,v) = v$. $c?x$ will send the data v through to processes SYN and $STOP$ through signals S_s and $S_{c?}$ separately. Finally, a token will be sent through S_{p1_end} to declare that P_1 has finished.

Regard $P_1 \parallel P_2$ as a whole, there are two input signals (S_{p1_start} and S_{p2_start}) and two output signals (S_{p1_end} and S_{p2_end}) in ForSyDe. Only input signals will affect the whole procedure of $P_1 \parallel P_2$, therefore, there are four kinds of possible combinations with the two inputs.

- *Combination 1* S_{p1_start} does not contain any token, but S_{p2_start} contains a token.
- *Combination 2* S_{p2_start} does not contain any token, but S_{p1_start} contains a token.
- *Combination 3* Neither S_{p1_start} nor S_{p2_start} contains any token.
- *Combination 4* Both of S_{p1_start} and S_{p2_start} contain tokens.

The behavior of signals are different according to four kinds of combinations of inputs, and transmitted tokens are shown in Table 4.8.

	Combination 1	*Combination 2*	*Combination 3*	*Combination 4*
S_{p1_start}	$<>$	$< s >$	$<>$	$< s >$
S_{p2_start}	$< s >$	$<>$	$<>$	$< s >$
S_v	$< 2 >$	$<>$	$<>$	$< 2 >$
S_s	$<>$	$<>$	$<>$	$< 2 >$
$S_{c?}$	$<>$	$<>$	$<>$	$< 2 >$
$S_{c!}$	$<>$	$<>$	$<>$	$< 2 >$
S_{p1_end}	$<>$	$<>$	$<>$	$< t >$
S_{p2_end}	$<>$	$<>$	$<>$	$< t >$

Table 4.1: The behavior of signals in $P_1 \parallel P_2$ ForSyDe model

As we known, every trace of the CSP process can be reconstructed from the signals in the corresponding ForSyDe model. Check the output signals $S_{c?}$ and $S_{c!}$, we will find that only Combination 4 can output an integer 2, the other three combinations cannot output any tokens. Related results of output to the

traces of CSP process $P_1 \parallel P_2$, it is showed that the first three combinations are related to trace <>, and the Combination 4 is related to the trace $< c : 2 >$.

4.9 Deterministic choice control

When there is a deterministic choice ($\Box$) in a CSP process, then this process is controlled by another CSP process or the environment. Some concrete examples are given below to show the how the parallel processes work. The choices in examples are binary choice with one channel in each branch.

CSP description:
$P_3 = (a?x- > STOP)\Box(b?y- > STOP)$
$P_4 = (a!3- > STOP)\Box(b!4- > STOP)$
$P_5 = a!5- > STOP$
where $\alpha(a) = \alpha(b) = Z$

P_3 contains a deterministic choice which is used to choose channel a or channel b to receive an integer, and then stops. P_4 matches P_3 perfectly, it also contains a deterministic choice to send an integer 3 through channel a or send an integer 4 through channel b. P_5 will send an integer 5 through channel a, and then stop.

We assume P_3 is in a Simple Network, and there are four different related CSP networks. The deterministic choice selection differs from different networks.

- *Network 1* $= P_3 \parallel P_4$

  If P_3 is in parallel with P_4, the performance of CSP process $P_3 \parallel P_4$ is to transmit 3 through channel a or 4 through channel b, then stop. In ForSyDe, the two processes P_3 and P_4 can be modeled in Figure 3.8 separately, and connect related signals inside P_3 and P_4 to fulfill pairwise communications of channel a and channel b, like in Figure 4.9. A ForSyDe model for $P_3 \parallel P_4$ is in Figure 4.10.

  Generally, in the entire thesis, signals in one figure with the same name are assumed to be connected so as to make ForSyDe framework more succinct. For example, in Figure 4.10, the output signal S_{a_v} of process $a!v$ is connected to the input signal S_{a_v} of process $a?x$, as they have the same names.

  In ForSyDe model, only use one choice signal T to control both choice operators in P_3 and P_4. In this way, both P_3 and P_4 can agree on the

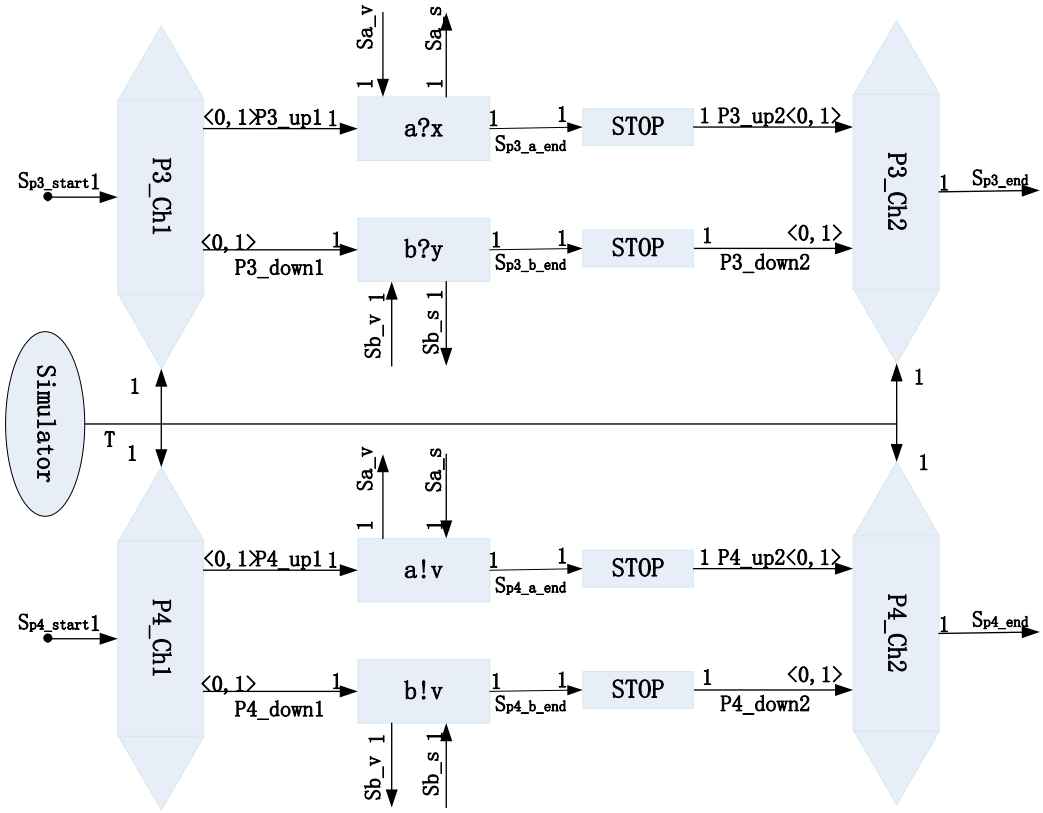


Figure 4.10: A ForSyDe model for $P_3 \parallel P_4$

upper signals or the lower signals to use. The way to control the signal T is similar to non-deterministic choice, an extra simulator is used outside this ForSyDe model to generate random data.

- *Network 2* = P_3

  If only P_3 inside CSP network contains channels a or b, this kind of CSP network must communicate with the environment. We assume the environment could perform any behavior in this thesis, so both channels a and b are available. The problem becomes the same as $Network1$, then the model for P_3 is the same as Figure 4.5.

- *Network 3* = $P_3 \parallel P_5$, and no communication with environment.

  If P_3 is in parallel with P_5 without any communication with environmnet, the performance of CSP process $P_3 \parallel P_5$ is always to transmit an integer 5 via channel a, and then stop. P_5 can be modeled similar to Figure 4.8 which performs like sending a value and stopping. The ForSyDe model for $P_3 \parallel P_5$ without environment communication is modeled in Figure 4.11.

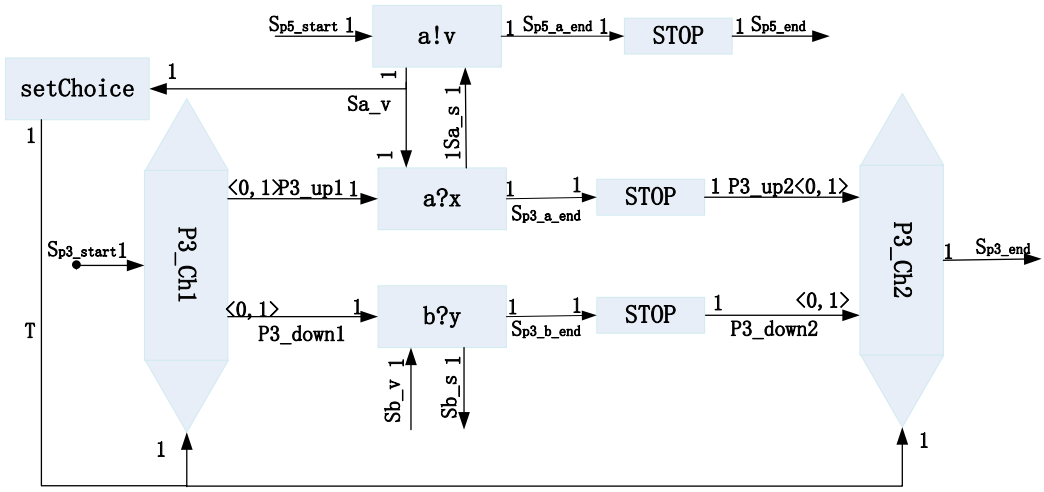


Figure 4.11: A ForSyDe model for $P_3 \parallel P_5$ without environment communication

When S_{p5_start} contains a token, it will send a token through S_{a_v}. This token can be splited into two, one is used for communicating with process $a?x$, and the other is connected to a process $setChoice$. Process $setChoice$ is a process which performs like that once receiving a token from signal

S_{a_v}, it will send a token through signal T to make a choice that choose the upper signals. Since no communication with environment, it will always choose the upper signals of the choice operators.

- *Network 4* $= P_3 \parallel P_5$, and communication with environment is allowed.

  Inside *Network 4*, only channel a can be chosen, but as we assumed before, the environment can perform any behavior. Therefore, P_3 can also get a token via channel b from the environment. Since both of the choices are available, it becomes the same problem as *Network 1* too. The model for $P_3 \parallel P_5$ is with environment communication is modeled in Figure 4.12. Signal T is connected to an extra simulator, to choose either communicating inside *Network 4* via channel a or communicating with environment via channel b.

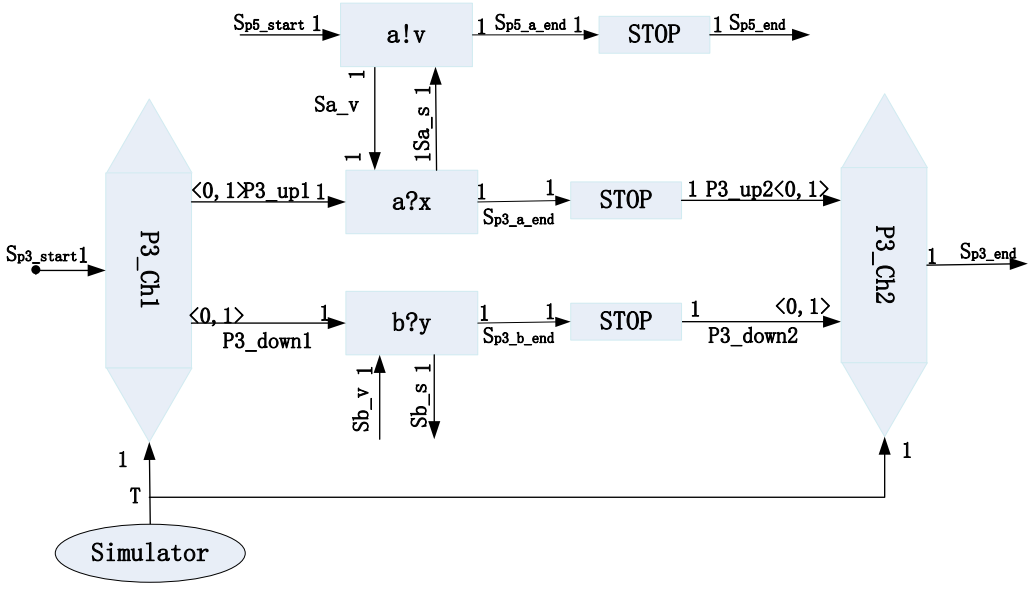


Figure 4.12: A ForSyDe model for $P_3 \parallel P_5$ with environment communication

Chapter 5

Construct ForSyDe models for Simple Network

In this chapter, we will generalize ForSyDe models for CSP processes. We will also give a general idea about constructing pairwise communicating concurrency in Simple Network and choice control frameworks.

5.1 General model

From a general point of view, every CSP process or event could be translated to a process with one start input signal (S_{start}), one procedural output signal (S_{end}), several environment control input signals ($\overrightarrow{S_{env\_in}}$) and environment control output signals ($\overrightarrow{S_{env\_out}}$). The environment control signals are connected to other processes. A general ForSyDe model is in Figure 5.1.

Figure 5.1: A general ForSyDe model

The main idea of translating a CSP process is to separate each internal event and process, translate every signal CSP event or primitive CSP process into a ForSyDe model, then connect related signals together to make a whole ForSyDe model.

In a CSP network, $P_1 \parallel P_2 \parallel P_3 \parallel ... \parallel P_n$, where $n \in [1, \infty)$, P_i (i∈[1,n]) can include primitive processes, channel events. It also could be a binary selective process or a recursive process. However, any kind of CSP process can be matched into model in Figure 5.1. Below, we will discuss some possible forms of P_i.

5.1.1 Primitive process

A primitive process (SKIP or STOP) in P_i has empty vectors $\overrightarrow{S_{env\_in}}$ and $\overrightarrow{S_{env\_out}}$. So only S_{start} and S_{end} signals exist to stand for a start input signal and a procedural output signal. A ForSyDe model for primitive process is in Figure 5.2.

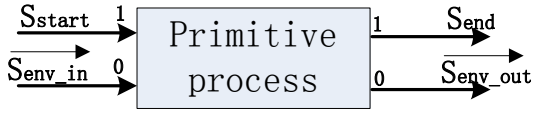


Figure 5.2: A ForSyDe model for a primitive process

5.1.2 Channel event

Channel events are related to sending or receiving events in a CSP process.

If $out(P_i)$ is not empty, it means CSP process P_i contains some sending events. For an arbitrary channel *[CHANNEL]* with any communitation data type *[DATA]*, the single sending event is related to a fixed ForSyDe model in Figure 5.3.

Inside process *CHANNEL!DATA*, it contains two internal processes $Exe\_CHANNEL!DATA$ and *SYN*. $Exe\_CHANNEL!DATA$ is the execution part of sending event, and it outputs an environment control signal to a related

Figure 5.3: A ForSyDe model for a single sending event

receiving part. *SYN* is used to wait for an environment control signal from receiving part for synchronization. It is illustrated in Figure 5.4.

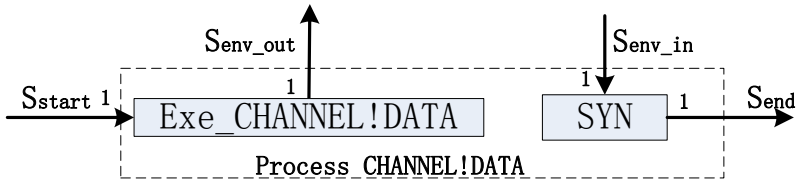


Figure 5.4: A ForSyDe model for a sending event with internal processes

If $in(P_i)$ is not empty, it means CSP process P_i contains some receiving events. For an arbitrary channel *[CHANNEL]* with any communitation data type *[DATA]*, a single receiving event is related to a ForSyDe model in Figure 5.5.

Figure 5.5: A ForSyDe model for a single receiving event

Therefore, for any P_i, if there is one channel in it, a pair of signals S_{env_in} and S_{env_out} will be added to the whole ForSyDe model of P_i. We assume c is a set of channels involved in P_i, so $c = in(P_i) \cup out(P_i)$. Figure 5.6 shows P_i with a set of channels c.

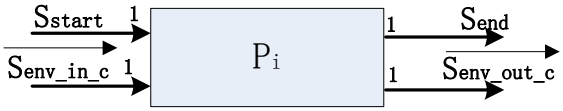


Figure 5.6: A ForSyDe model for P_i with a set of channels c

5.1.3 Binary selective process

In CSP, two choice branches of binary selective process are separated by the notation ($\sqcap$ or $\Box$). We need to separate one branch from another, and get two independent processes. The two processes are any arbitrary CSP processes X and Y, and can be generally modeled in Figure 5.7.

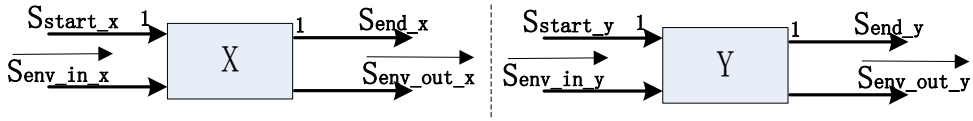


Figure 5.7: ForSyDe models for processes X and Y

In the ForSyDe model for a CSP choice operator, there are two upper signals (up1 and up2) and two lower signals (down1 and down2) in Figure 3.8. Connect up1 with S_{start_x} and up2 with S_{end_x} of process X, and connect down1 with S_{start_y} and down2 with S_{end_y} of process Y. Therefore, the choice between processes X and Y is modeled in Figure 5.8.

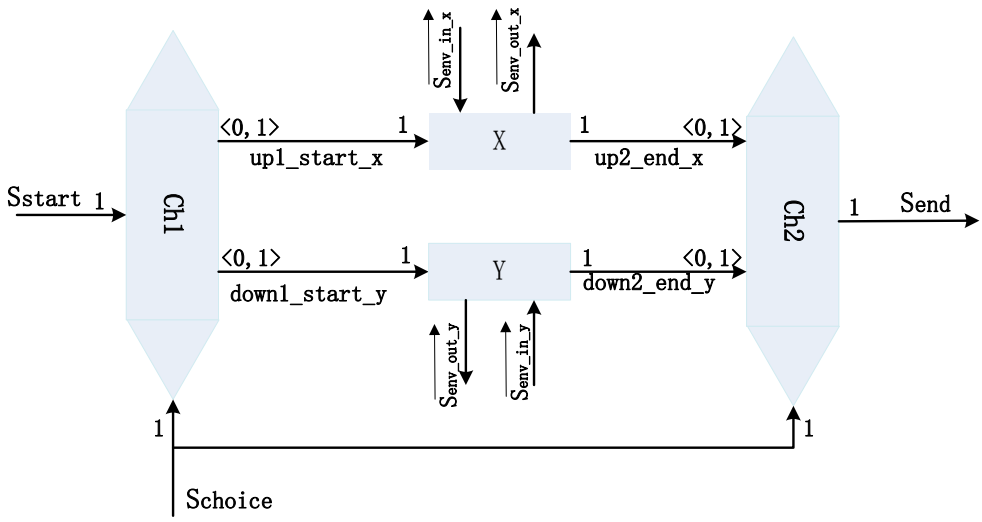


Figure 5.8: A ForSyDe model for a choice between X and Y

Regardless internal signals, an abstract ForSyDe model for a binary selective CSP process between two arbitrary processes X and Y is in Figure 5.9. $S_{env\_in\_choice}$ is also an environment control input signal, which decides process X or Y to be chosen.

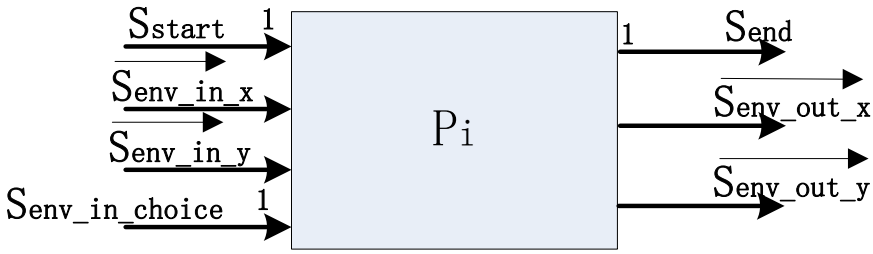


Figure 5.9: An abstract ForSyDe model for a choice bewteen X and Y

5.1.4 Recursive process

Recursion happens when a CSP process P_i contains itself in its own definition. In the definition of a guarded recursive process, the right-hand side of the equation is always after at least one event. In ForSyDe, we know that any CSP process P_i can be modeled to general model in Figure 5.1.

The way to describe recursion in ForSyDe is using a circle with a Delay Process. The input signal of the Delay Process is signal S_{end} of P_i, and the output signal of the Delay Process is connected to the very beginning process in P_i as another input signal of that process. So if P_i contains a recursion, it can be modeled in Figure 5.10.

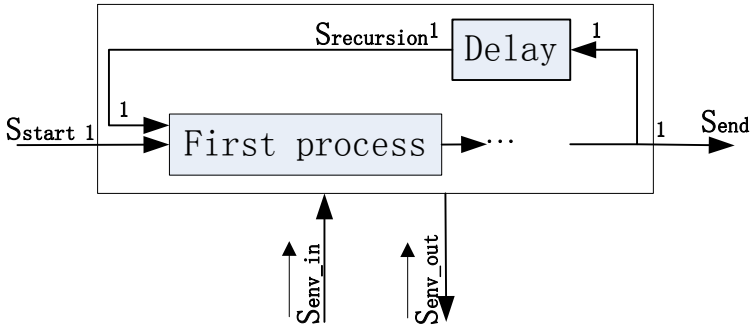


Figure 5.10: A ForSyDe model for a recursive process P_i

There is a problem when the model for P_i begins with a ForSyDe choice operator, because the choice operator in ForSyDe only allows one input signal. In this case, we can add one process *Help* before the choice operator to handle this issue, which is modeled in Figure 5.11.

The process *Help* is used to get one input signal to start the whole process P_i, as well as another signal from Delay processes, and then forward tokens from the start signal to its output.

Thus, $Help(x, y) = x$
where, x represents a token from S_{start}; y represents a token from $S_{recursion}$

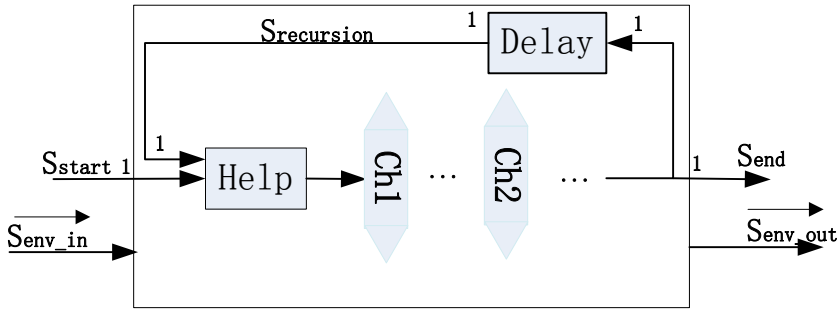


Figure 5.11: A ForSyDe model for a recursive process P_i begins with a choice operator

5.2 Concrete procedures

For any P_i in CSP network, $Network = P_1 \parallel P_2 \parallel P_3 \parallel ... \parallel P_n, i \in [1, n]$, the procedures of translating P_i is as follow:

- Step 1 Analyze the CSP description to get sending and receiving channels involved and the number of choice operators, in order to determine the related environment control signals. It also needs to check whether it contains P_i itself.
- Step 2 Translate the CSP process from beginning. If the part is an event or a process except a binary selective process with form ($X \sqcap Y$ or $X \Box Y$), we can translate it into a ForSyDe model in Figure 5.1 directly. If it is a binary selective process, use a ForSyDe model with choice operators in Figure 5.8 to separate processes. For every separated process, repeat Step 2 until reach the end. During translation, if reaches P_i itself, just ignore it, and leave it later.
- Step 3 Once getting to the end, if P_i is a recursive CSP process, add a Delay Process to the end of P_i, make a circle back to the first process, and details are in Sect. 5.1.4.

Now, follow the procedures above, there is an example to show how to translate a CSP process P_i:

$P_i = a?x \rightarrow (b?y \rightarrow P_i) \Box (c?z \rightarrow STOP)$

Steps are described as follows with related figures, in which we ignore names of signals and index beside signals.

1. Analyze the whole P_i, find $in(P_i) = \{a, b, c\}$, and one choice operator in CSP definition. The definition contains P_i itself, so it is a recursive process.

2. Start from the first event $a?x$, translate to a ForSyDe process $a?x$.

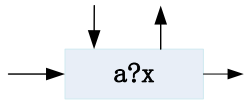


Figure 5.12: Step 2: Translating $a?x$

3. The rest of P_i is a binary selective process $(b?y \rightarrow P_i)\Box(c?z \rightarrow STOP)$, a pair of choice operators in ForSyDe is added following ForSyDe process $a?x$.

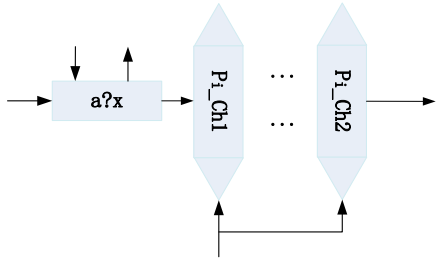


Figure 5.13: Step 3: Add a pair of choice operators

4. One branch of the binary selective process is $b?y \rightarrow P_i$, construct ForSyDe model for $b?y$, and ignore P_i at this moment. Put the framework of $b?y$ to the upper position in the pair of choice operators.

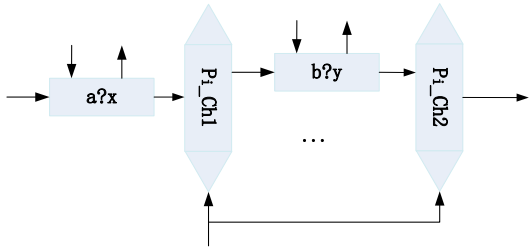


Figure 5.14: Step 4: Add a choice branch $b?y \rightarrow P_i$

5. The other branch of the binary selective process is $c?z \rightarrow STOP$, construct ForSyDe model for this branch and add it to the lower position in the pair of choice operators.

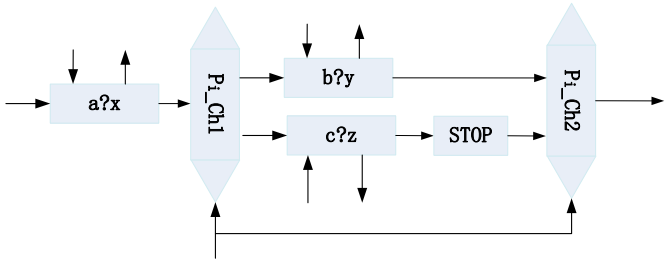


Figure 5.15: Step 5: Add another choice branch $c?z \rightarrow STOP$

6. It has already reached the end of CSP description, and P_i is a recursive process, so a Delay Process should be add to the end, which with an output signal connected to the process $a?x$.

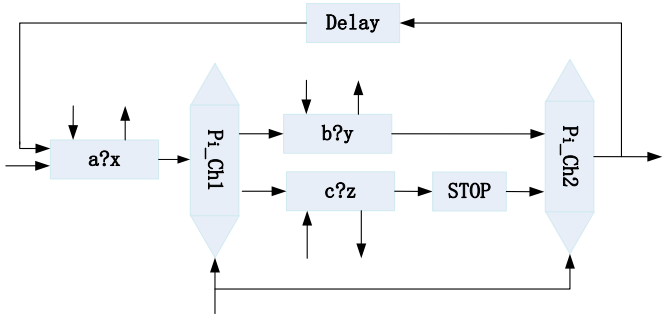


Figure 5.16: Step 6: Add a Delay Process

5.3 Pairwise communicating concurrency

Pairwise communication in a CSP network means one sender has only one fixed receiver partner to communicate. The concurrency of pairwise communication is to connect the environment control signals of both sending and receiving sides to make sure data transmission can take place at the same time. In Sect. 5.1.2 we have general sending and receiving models for arbitrary channel *[CHANNEL]*

with any communication data type *[DATA]*. After connecting S_{env_out} of *CHANNEL!DATA* to S_{env_in} of *CHANNEL?DATA*, and S_{env_in} of *CHANNEL!DATA* to S_{env_out} of *CHANNEL?DATA*, we will get the pairwise communication model in Figure 5.17.

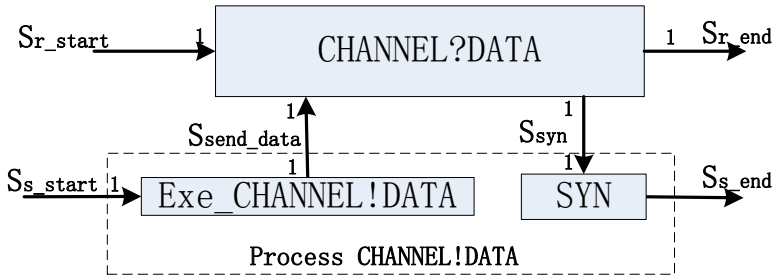


Figure 5.17: A general ForSyDe model for pairwise communication

The connected environment control signals are:

- S_{send_data}: it is from sending process to receiving process, and transmits a valid data for communication.
- S_{syn}: it is from receiving process to sending process, and sends the received data back to sending part to guarantee synchronization of both sides.

5.4 Choice control

As we know, any binary selective CSP process can be modeled as Figure 5.9. The way to control the selection is to find a corresponding signal to connect with $S_{env\_in\_choice}$. For any process P_i which binary choice with any arbitrary processes X and Y, it can be written as $P_i = X \sqcap Y$ or $P_i = X \Box Y$. We assume X' is a process which has the same communication channels as X, and $in(X) = out(X'), out(X) = in(X')$. For binary choice control of P_i, we can conclude two kinds of models.

Model One called *Simulator Model*, as there is an extra simulator exists. It is shown in Figure 5.18, and this model is related to three cases.

The first case is non-deterministic choice between X and Y ($X \sqcap Y$). Signal $S_{env\_in\_choice}$ is connected to an extra simulator outside ForSyDe model. This

simulator can generate tokens through $S_{env\_in\_choice}$ randomly.

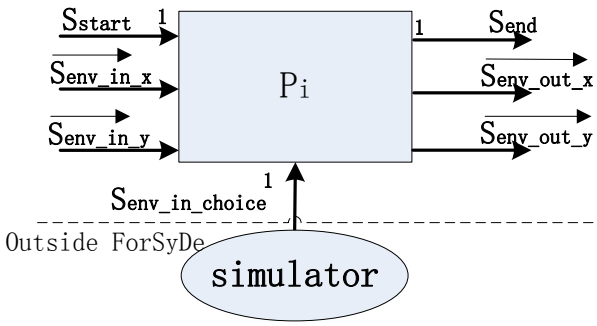


Figure 5.18: Simulator Model for choice control

The second case is that $P_i = X \square Y$. If a P_j ($P_j \neq P_i$) exists in the CSP network, and $P_j = X' \square Y'$, the ForSyDe model For $P_i \parallel P_j$ is in Figure 5.19. The environment control signals for channels in X and Y can be matched perfectly between ForSyDe model P_i and P_j. An extra simulator is connected to P_i and P_j with the same signal $S_{env\_in\_choice}$, in order to control choices in both processes concurrently. If we only take P_i or P_j point of view, the process is under Simulator Model separately.

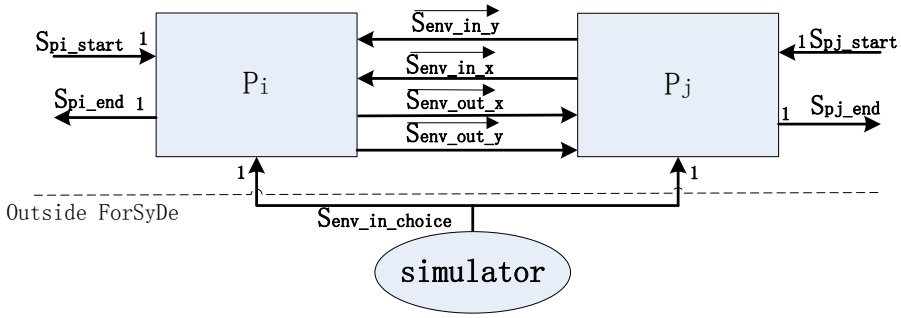


Figure 5.19: A ForSyDe model for $P_i \parallel P_j$

The third case is also for deterministic choice $P_i = X \square Y$, but X' and Y' could be only one or none of them exists in CSP network. However, this CSP network can communicate with environment, and we assume the environment can perform any behavior, so there could be X' or Y' in the environment. As a result, both choices of P_i are available, and an extra simulator is used to choose whether X or Y to be executed.

Model Two is called *Process Model*, since an extra process is needed to determine the choice. It is illustrated in Figure 5.20.

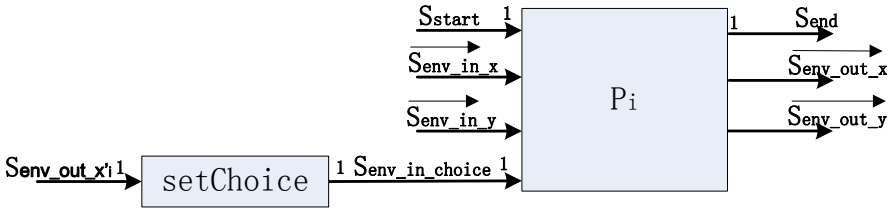


Figure 5.20: Process Model for a choice control with a selection of X

This model is only related to one case that $P_i = X \square Y$, and no communication with environment is allowed. Inside CSP network, only channels in one of the two choice processes (X and Y) have their communication partners in the rest of processes. We assume X' which is modeled in Figure 5.21 is inside the CSP network, but Y' is not in CSP network.

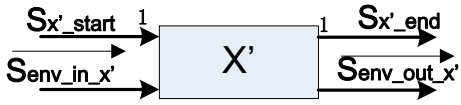


Figure 5.21: A ForSyDe model for X'

Signal $S_{env\_out\_x'i}$ in Figure 5.20 is one of $\overrightarrow{S_{env\_out\_x'}}$ in Figure 5.21, once process *setChoice* gets a token from $S_{env\_out\_x'i}$, it will send a token through signal $S_{env\_in\_choice}$ to choose process X for P_i.

Chapter 6

Advanced Network Communication

The communication between two processes we have discussed in the previous chapters is based on *Simple Networks*. However, the CSP communication could under *Advanced Networks*. We will introduce a new operator to resolve communication under *Advanced Networks* in this chapter.

6.1 A shared channel

First of all, let's come to an example under an *Advanced Network*, which is described below.

$Network1 = P_1 \parallel P_2 \parallel P_3$
$P_1 = c?x \rightarrow d!4 \rightarrow c?y \rightarrow P_1$
$P_2 = c!5 \rightarrow c!6 \rightarrow c!7 \rightarrow P_2$
$P_3 = d?y \rightarrow P_3$
where $\alpha(c) = \alpha(d) = Z$

We can find that channel d only has one occurrence of the sending event $d!4$ in P_1, and one occurrence of receiving event $d?y$ in P_3, so the communication

between them could be related to the ForSyDe model in Figure 5.17. However, towards channel c, there are three sending events in P_2 and two receiving events in P_1. We can find several sending and receiving behaviors could be performed on the same channel in an Advanced Network, and such kind of channel is called as a *shared channel* in this thesis. Since both P_1 and P_2 contain a recursion, there is no fixed collocation between one sending event and one receiving event through a shared channel c.

6.2 Selector on a shared channel

In above CSP description, at most one sending event is ready at a time, due to the sequential structure of processes, and so as receiving events. However, we need to find which sending event and receiving event are supposed to communicate.

A pair of selectors (*Selector*, *Selector'*) is proposed to resolve the problem, a structure of selector mechanism on channel c shown in Figure 6.1. We will find that *Selector* and *Selector'* of channel c are illustrated in the middle of sending and receiving parts. For the *Selector* in Figure 6.1, there is an input signal connected with a sending event, a pair of input and output signals related to a receiving event, while the output signal of the pair is connected to the receiving event. On the other hand, the *Selector'* has an input signal coming from an receiving event, a pair of input and output signals related to a sending event, while the output signal of the pair is connected to the sending event.

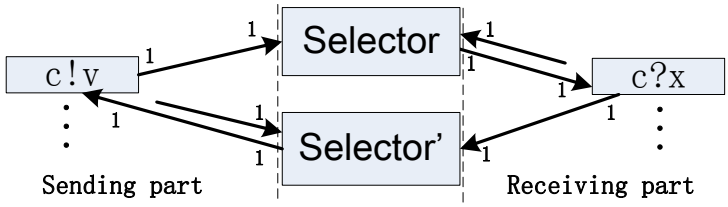


Figure 6.1: A selector mechanism on channel c

First, we will take a look at the upper half of Figure 6.1, only consider about the *Selector* and processes which are connected to the *Selector*. So for a CSP network *Network1* in Sect. 6.1, we can model a ForSyDe framework for P_1 and P_2 in Figure 6.2, in the ForSyDe framework, we can find that the sending part is corresponding to P_2, and receiving part is corresponding to P_1.

Let's take sending and receiving part separately. From sending part point of

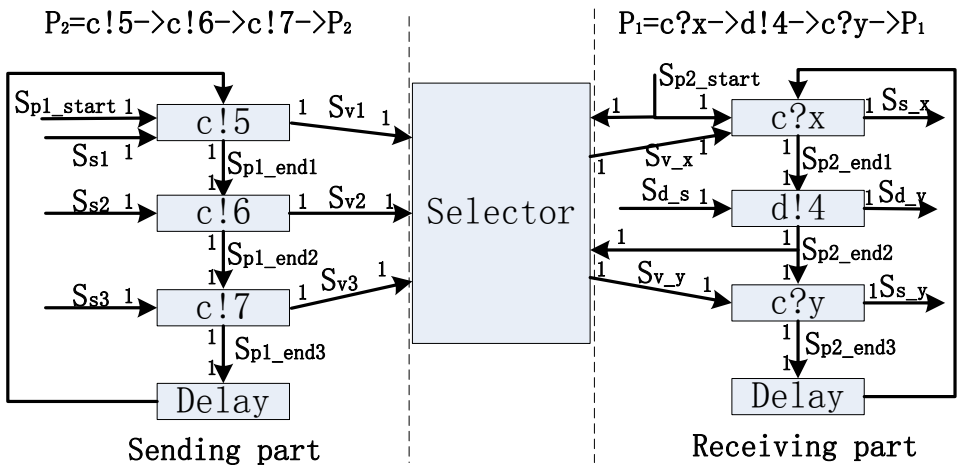


Figure 6.2: A ForSyDe model for the example with the *Selector* on channel c

view, each of three senders of channel c has an environment control signal to connect to *Selector* as three input signals of the *Selector*. However, among the three input signals, only one of them can send a token carrying a valid data at a time.

Towards the receiving part, there is a pair of input and output signals and each pair is related to one receiver. For each pair, the input signal of the *Selector* is a branch from a start signal of a receiver, and the output signal of the *Selector* is connected with an environment control input signal from that receiver. For instance, signals S_{p2_start} and S_{v_x} in Figure 6.2 are in a pair which is related to the receiver process $c?x$. After getting a token from its sending part, the *Selector* will check all of input signals from the receiving part. If there is one of input signals contains a token, and for sure this input signal belongs to a pair, the *Selector* will forward the token to its output signal of that pair. If no token exists in any input signal from its receiving part, the *Selector* will wait and keep the data until one of its input signals from receiving part has a token.

When a receiver has got a token, it needs to send a token back to the sender who starts the communication so as to accomplish the entire communication. So the lower half of Figure 6.1 with the *Selector'* is to achieve this function. Regardless non-relevant signals and processes with channel c, the other half with *Selector'* is modeled in Figure 6.3. We can find that the *Selector'* has two input signals connected to receivers and three pairs of input and output signals related to senders. The functionality of the *Selector'* is the same as the *Selector*.

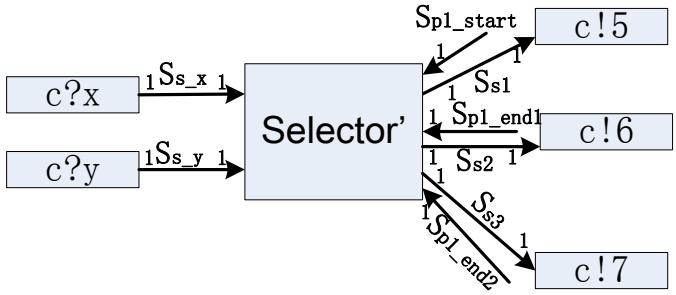


Figure 6.3: A ForSyDe model for the example with the *Selector′* on channel *c*

Notice that signals with the same name in Figure 6.2 and 6.3 are supposed to be the same signals.

The whole procedure of data communication on a shared channel is that when one of senders sends a token to the *Selector*, the *Selector* will forward the token to only one receiver who prepares for receiving. If no receiver is ready to get the data, it will keep the data inside *Selector* until one receiver is ready. After the receiver finishing receiving, the receiver will send a token to the *Selector′*, the *Selector′* can also forward the token to the sender who starts this communication and waits for a reply, unlike *Selector*, there always be a sender who is ready to get this feedback token.

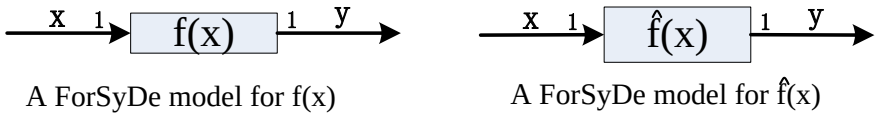
6.3 Functions in selector

From the above example, we can see that both *Selector* and *Selector′* have the same functionality, and we call such kind of ForSyDe model as *selector* model, which is consist of several input signals from a sending part, and several pairs of input and output signals from a receiving part. Below, let's take a look at the general procedure of the *selector* model.

In a *selector* model, it will check all the input signals from the sending part for every time unit, if there is no tokens at that moment, and no tokens are produced for output. If a token appears at one input signal from the sending part, check all the input signals from the receiving part to find which receiver can receive this token, and then forward this token to that related output signal. If none of receivers is ready for getting that token, the token will be stored in *selector* until one receiver prepares for taking it. Notice that the model only allows one sender and one receiver are activated, thus, if one of senders or receivers does not finish

communicating, other senders or receivers cannot prepare for executing.

We shall now describe details of how a *selector* can be modeled in ForSyDe. First, let's have a look at two functions $f(x)$, $\hat{f}(x)$, which are modeled separately in Figure 6.4.



A ForSyDe model for f(x) A ForSyDe model for f̂(x)

Figure 6.4: ForSyDe models for $f(x)$ and $\hat{f}(x)$

Process $f(x)$:
Input signal $x \in A$, A is a valid data set by a certain definition, such as integer, float.
Output signal $y \in B$, B is also a valid data set by a certain definition, and $y = f(x)$.
f(x) is a one-to-one map, to fulfill $A \rightarrow B$.

Process $\hat{f}(x)$:
Input signal $x \in \hat{A}$, $\hat{A} = \{None\} \cup A$.
Output signal $y \in \hat{B}$, $\hat{B} = \{None\} \cup B$, and $y = \hat{f}(x)$.
Here, ***None*** represents the signal x does not contain any tokens.

The definition for $\hat{f}(x)$ is:

$$\hat{f}(x) = \begin{cases} None, & x = None \\ f(x), & x \in A \end{cases}$$

In ForSyDe, process $\hat{f}(x)$ will check signal x for every time unit. At a certain time unit, if no token appears, no token is sent out through y, and if there is a token a via signal x, $a \in A$, it will send a related token *f(a)* through signal y.

Now, we extends the function $\hat{f}(x)$ with several input signals ($\overrightarrow{x}$) connected with senders and several pairs of input and output signals ($\overrightarrow{start\_y}$ and $\overrightarrow{y}$) related to receivers to model the *selector*. A general ForSyDe model for *selector* is in Figure 6.5.

For all $x_i \in \overrightarrow{x}, x_i \in \hat{A}$, since at most one of the input signals from the sending part can contain a token at a certain time unit, the function selector($\overrightarrow{x}$) is defined as:

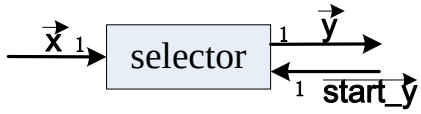


Figure 6.5: A ForSyDe model for *selector*

$$selector(\overrightarrow{x}) = \begin{cases} None, & any\ x_i \in \overrightarrow{x}, x_i = None \\ f(x_i) = x_i, & x_i, x_j \in \overrightarrow{x};\ any\ x_j \neq x_i, x_i \in A, x_j = None \end{cases}$$

From the equation of function $selector(\overrightarrow{x})$, two results can be reached.

- If there is no input signal contains a token, $selector(\overrightarrow{x}) = None$, all the output signals $\overrightarrow{y}$ will not get any tokens.
- If just one input signal x_i is available, $selector(\overrightarrow{x}) = x_i$. Only one signal y_j ($y_j \in \overrightarrow{y}$) with a related signal $start\_y_j$ ($start\_y_j \in \overrightarrow{start\_y}$) containing a token could get the token with the value x_i, and any other signal y_k ($y_k \neq y_j, y_k \in \overrightarrow{y}$) will not receive any token. However, if no $start\_y_j$ has a token at that moment, the token will be kept inside *selector*, until one $start\_y_j$ signal contains a token.

A way to store a token inside *selector* is discussed here. We can introduce a Delay Process connected with *selector*, shown in Figure 6.6. When the *selector* has got a value from $\overrightarrow{x}$ but none of $\overrightarrow{start\_y}$ has a token, the value could be sent to the Delay Process first. Then at the next time unit, this value will be sent back to the *selector*, and at that moment the *selector* will check $\overrightarrow{start\_y}$ again to see whether the value could be sent out.

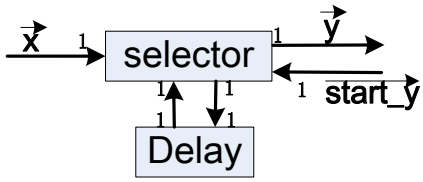


Figure 6.6: A ForSyDe model for *selector* with a Delay Process

6.4 An example with selector

A simple example is in Figure 6.7, which illustrates a model containing a *selector* with two senders and two receivers.

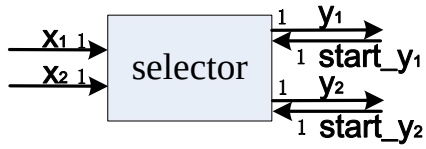


Figure 6.7: An example with *selector*

Since x_1 and x_2 cannot contain tokens at the same time, and so as $start\_y_1$ and $start\_y_2$, the possible combinations of input signals from sending part are given in Table 6.1. **v** in table represents a valid data from a sender. The combinations of $start\_y_1$ and $start\_y_2$ are collected in Table 6.2, where **s** represents a valid token.

Combination	x_1	x_2	Meaning
Send_1	None	None	No senders contain a value
Send_2	v	None	x_1 contains a token, v
Send_3	None	v	x_2 contains a token, v

Table 6.1: Possible combinations of x_1 *and* x_2

Combination	$start\_y_1$	$start\_y_2$	Meaning
Receive_1	None	None	No receivers are ready to get a token
Receive_2	s	None	y_1 is ready to get a token
Receive_3	None	s	y_2 is ready to get a token

Table 6.2: Possible combinations of $start\_y_1$ and $start\_y_2$

Combine the Send_i with Receive_j, $i, j \in \{1, 2, 3\}$, we find get different output signal combinations through y_1 and y_2. Mathmatically, there are nine combinations between Send_i and Receive_j, however, only three results may take place. The results and their related sending and receiving combinations are shown in Table 6.3.

Results	Combinations
Neither y_1 nor y_2 can get a token	Send_1×Receive_j, Send_i×Receive_1, $(i, j \in \{1, 2, 3\})$
y_1 can get a token v	Send_2×Receive_2, Send_3×Receive_2
y_2 can get a token v	Send_2×Receive_3, Send_3×Receive_3

Table 6.3: Results and related input combinations

From the results, we will find that only if Send_1 or Receive_1 take place, there is no token to transmit out. When Send_2 or Send_3 happens, y_1 may output a token if Receive_2 takes place, and y_2 may output a token if Receive_3 takes place.

Chapter 7

An alternative approach

Besides translation from CSP to ForSyDe model, an alternative approach will be discussed briefly in this chapter. CSP processes are translated to a *task graph*, and then schedule a task graph on multi-core platforms.

7.1 Task Graph

A task graph [8] is a task scheduling area, where the nodes represent the tasks and the edges represent the communications between the tasks. Scheduling a task graph onto multi-core platforms with several processors is a trade-off between maximizing concurrency and minimizing interprocessor communication costs. An example of task graph is illustrated in Figure 7.1.

This example contains five tasks: T1, T2, T3, T4 and T5. The execution time of each task is assigned beside the node, such as the execution time of T3 is 3 time units. The communication time between two tasks is assigned next to a related arrow, just like one time unit is the communication time cost between T1 and T3.

We have to notice that the communication time between two connected tasks

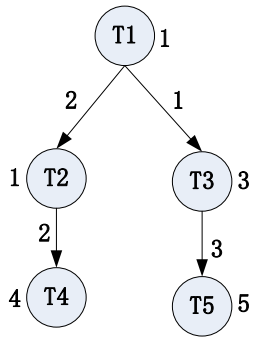


Figure 7.1: A sample for task graph

in task graph is only taken into account when the two tasks are scheduled on different processors, so if the two connected tasks are scheduled on the same processor, the communication time could be ignored.

Below, we introduce a very realistic example to show how the translation and scheduling after translation work.

7.2 Example based mapping CSP to task graph

7.2.1 VM system overview

VM system in this chapter is a system to serve coffee as well as chocolate, which is more complicated than VM in Sect. 2.4.2. A customer can insert a coin to choose *chocolate* or *coffee*. If chocolate is needed, the machine will make a cup of chocolate after well. If coffee is chosen, the machine will crush coffee beans, after that it will grow coffee.

The paths of serving chocolate and coffee are independent, as a result, VM can take the order of chocolate and coffee one after another without any conflict. When a customer orders a chocolate, it takes some time to make. So during the processing time, the VM can accept another order of chocolate, but the new order is pending until the last cup of chocolate is finished. Towards serving coffee, during the processing of crushing beans, the VM will accept a new order of coffee as a pending order until VM finishes crushing beans and starts to grow coffee. If there is a pending order in this system, VM cannot accept another order, regardless the same drink or not.

7.2.2 CSP description for VM

Events
This simple VM includes six events as follows, three external events from customers and three internal events.

External events:

- coin: the insertion of a coin in the slot
- choc: the selection of ordering a cup of chocolate
- coffee: the selection of ordering a cup of coffee

Internal events:

- makingchoc: the procedure of extracting a chocolate from the dispenser
- crushing: the procedure of crushing coffee beans
- growing: the growth of coffee from the dispenser

Processes
Process VM is consist of four processes, which sending or receiving messages between each other. The definition of four processes is:

- order: order one drink
- makechoc: make a cup of chocolate
- crushbean: crush coffee beans
- growcoffee: grow a cup of coffee

Channels
The channels between different processes are defined below:

- ch: a channel from order to makechoc
- bean: a channel from order to crushbean
- gr: a channel from crushbean to growcoffee

Only one message {start} is transfered in these three channels, after sending or receiving this message, the process can move on. The frame of processes and channels are shown in Figure 7.2.

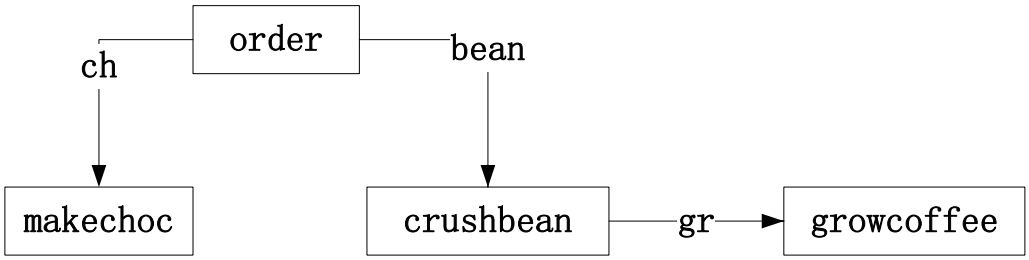


Figure 7.2: frame of processes and channels

CSP description

$order = coin \rightarrow (choc \rightarrow ch!\{start\} \rightarrow order)$

$\Box(coffee \rightarrow bean!\{start\} \rightarrow order)$

$makechoc = ch?\{start\} \rightarrow makingchoc \rightarrow makechoc$

$crushbean = bean?\{start\} \rightarrow crushing \rightarrow gr!\{start\} \rightarrow crushbean$

$growcoffee = gr?\{start\} \rightarrow growing \rightarrow growcoffee$

$VM = order \parallel makechoc \parallel crushbean \parallel growcoffee$

7.2.3 Task graph

Each event in CSP can be mapped to a node in task graph, and assigned a reasonable execution time, which is not mentioned in CSP. Task nodes for events involved in VM system and their execution time are shown in table 7.1.

Events	coin	choc	coffee	makingchoc	crushing	growing
Task node	T1	T2	T3	T4	T5	T6
Execution Time	1	1	1	4	4	3

Table 7.1: nodes and execution time

The translation of some notations are discussed below.

The prefix notation between two events in CSP is related to an arrow between two related nodes in task graph. For example, event *coin* is a prefix of *choc*, so there is an arrow from task node T1 to T2.

Communication channels in CSP are also mapped to an arrow between two related nodes. The arrow starts from a node, which is the prefix of the sending part of the channel, and it ends up with a node, which is the next event or the next primitive process of the receiving part of the channel. For instance, channel *gr* has event *crushing* as its prefix of the sending part $gr!\{start\}$, and event *growing* is its successor of the receiving part $gr?\{start\}$. So an arrow which starts from T5 and ends up with T6 is related.

The way to represent recursion is an arrow which starts from the related last node and points back to the first node. If there is only one event involved in a process, such as *makechoc*, this arrow goes back to the only task node $T4$.

Choice in task graph cannot tell the differences between deterministic and non-deterministic choices. When coming to a selective process, use many arrows starting from the same node, but ending up with different nodes to represent more branches. In process *order*, after event *coin*, two events can be chosen, as a result, two arrows both starting from node T1, but going to T2 and T3 separately are related.

Task graph of VM is illustrated in Figure 7.3. Left side of task graph (T1, T2, T4) is serving chocolate, while right side (T1, T3, T5, T6) is serving coffee.

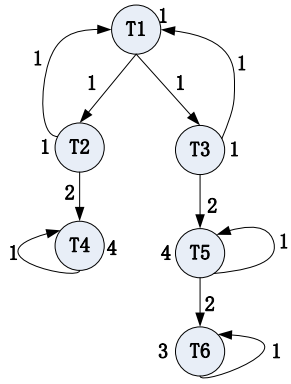


Figure 7.3: task graph for VM

7.2.4 Executing on multi-core platforms

Single-core platform contains one processor, in this case, it doesn't need to consider about the communication time in task graph. However, it has to execute task nodes one by one, some concurrent tasks cannot show their advantages. For instance, the VM gets two orders, one is for coffee, and the other is for chocolate. Task scheduling in processor P is shown in Figure 7.4.

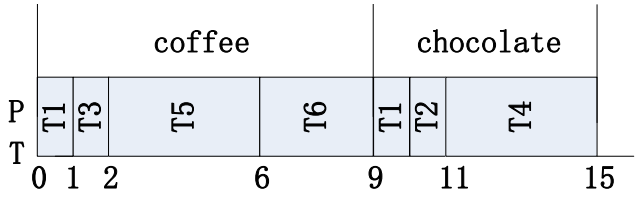


Figure 7.4: task scheduling in single processor

We can find VM serves coffee and chocolate sequentially since only one processor exists, and it costs *15* time units to finish serving. For single processor, it consumes *9* time units to serve coffee, and *6* time units to serve chocolate. If order n cups of coffee and m cups of chocolate, the total time is to sum up the individual processing time, as a result, *(9n+6m)* time units are needed.

Multi-core platform contains at least two processors and links between processors. In VM system, if we order one coffee and one chocolate in a platform with two processors with the architecture in Figure 7.5. It is scheduled as in Figure 7.6.

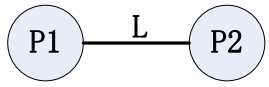


Figure 7.5: topology graph with two processors

In Figure 7.6, all tasks have finished being scheduled at time *9*, it saves 40% time compare with single processor. Processor P1 executes tasks serving coffee, while P2 executes tasks serving chocolate. L is the link between P1 and P2, and communication time from T3 to T1 is scheduled on it.

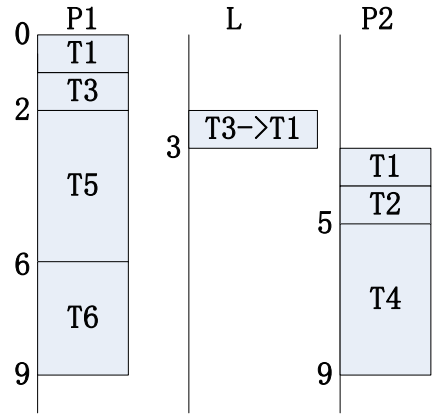


Figure 7.6: one coffee and one chocolate scheduled on two processors

We can also introduce three processors platform like in Figure 7.7, and schedule more than two orders, e.g. one coffee and two chocolate orders, which are scheduled in Figure 7.8.

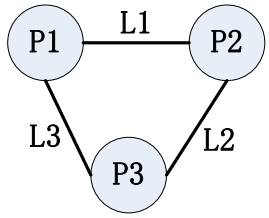


Figure 7.7: topology graph with three processors

Figure 7.8 shows P1 serves one coffee, and P2 serves one chocolate as in Figure 7.6. After ordering one chocolate, VM can also accept another order of chocolate, which is executed on P3. However, the later order of chocolate has to wait until T4 in P2 has finished, then starts a new T4 in P3 at time *9*. The total time consumption is *13* time units, while in single platform it needs *21* time units.

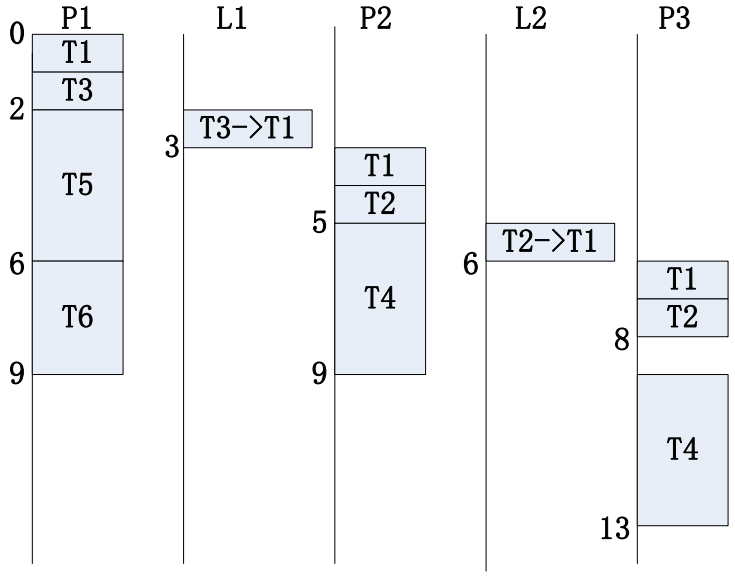


Figure 7.8: one coffee and two chocolates scheduled on three processors

Chapter 8

Conclusion

In this project we study the translation of a high-level model to a system-level model. The main part of this document illustrates what CSP and ForSyDe models are, and how to translate some subsets of CSP to ForSyDe models.

There are two reasons for this:

- CSP adds significantly to the expressiveness and abstractness which can be used in connection with ForSyDe.
- The translation shows how CSP can be executed on a platform with massive parallelism.

We have used some concrete examples first to show how the translation works, begin with simple CSP operators without concurrency, and then described how to extend these simple components to a concurrent mechanism. The concurrency discussed in this document covers communication concurrency under both *Simple Network* and *Advanced Network*. We have also discussed about controlling a choice selection under a parallel platform. After elaborating example based translations, we have generalized a ForSyDe framework for an arbitrary CSP process and some events. We've concluded general procedures of translating a CSP process, a way to construct pairwise communicating concurrency and a solution to control a choice selection as well.

We have also tried an alternative approach by translating CSP to task graphs. Task graph is a low-level model as well, since every single task is supposed to be scheduled on single or multi-core platforms. This approach does not appear so positive as events in CSP will be scattered after being translated to independent tasks, and it is not clear to see the interactions among processes in CSP. Besides, the concurrency expressed there is a concurrency due to several processors on multi-core platforms, not concurrent processes in CSP. However, we can represent a CSP process by a hierarchical process in ForSyDe, and any process in ForSyDe can be parallel to other processes, so interactions among such ForSyDe processes can show how the concurrency performs. As a result, from the expression of concurrency point of view, ForSyDe has its advantage on this translation CSP to a low-level model, compared with task graph.

We have achieved two main goals proposed at the beginning, although there may be some leakages during translation. We succeed to translate some subsets of CSP, such as primitive process, prefix, recursion, pairwise communication, and binary choice, to ForSyDe models. We also solve communication concurrency, and choice control problem in ForSyDe. Besides, this thesis could be a supplementary material for ForSyDe.

However, there are still many aspects we could improve. For example, when translating communication concurrency, we only consider about the situation that one sender is connected with one receiver. What if there are more receivers but only one sender is related? Signals for sending data could split to more branches without any problems, but only one environment control input signal of the sender is not enough. In this case, we may solve it by changing the model or duplicating the sender. Another issue is that, now only binary choice is considered in one process, but it could be more. One idea to figure it out is to combine more choice operators in ForSyDe together. One more difficult issue is if processes have common non-channel events, no evidence to show which one should take place earlier, unlike communication on channels. The solution of this issue is still hung.

Therefore, we could solve above issues first in the next research stage. Then consider about implementation of an application to display CSP on top of ForSyDe model, since all of the translations of models are discussed theoretically, we need an implementation to check whether it could work well.

During the research, we also find some shortcomings of ForSyDe, as follows:

- Document about ForSyDe is quite poor, the most knowledge we based on is from [7]. Expressiveness from the limited literature does not give us a clear definition and usage of some ForSyDe components.

- There are some limitations in connection with software design. It seems that ForSyDe lacks an abstract mechanism, for example, if we need a *selector* like in Figure 6.5, we have to invent it from very basic ForSyDe principles.

  Inspired by the definition of *selector* (Sect. 6.2) in CSP translation, it would be interesting to add a notion of a component to ForSyDe and to define a rich set of connectors for composing components. Such concept would enhance the applicability of ForSyDe significantly.

Appendix A

Wireless Sensor Networks

In appendix, we describe a CSP network of wireless sensor network, which was supposed to be translated to a ForSyDe framework when starting this thesis. Due to this CSP network is quite complicated and our skills of translation is not mature enough, we have to give it up. However, we hope such translation from complicated CSP networks to ForSyDe models can be achieved in the future.

A.1 Overview

Wireless sensor network (WSN) [10] is a network which is consist of an arbitrary number of nodes and one sink, shown in Figure A.1. Each node has its own identical number to correspond, and there are two major functions in nodes:

- To collect and produce data from its physical environment.
- To route data from itself and neighboring nodes towards a basic sink which collects all data produced by the WSN for further processing.

The routing algorithm of every node is to find the shortest distance from itself to the sink, and then send data to its neighboring node which is on the shortest

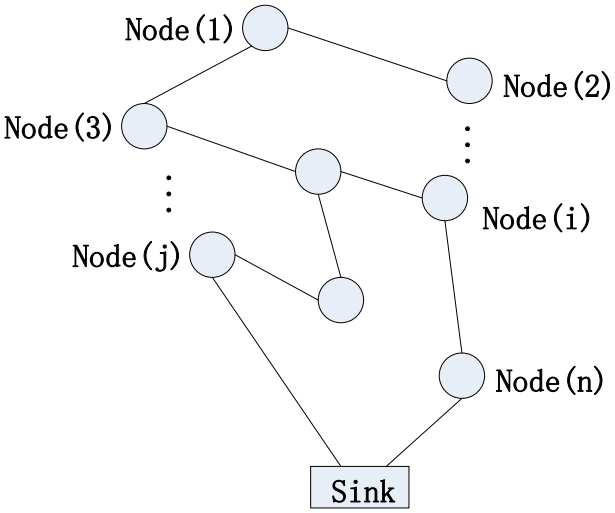


Figure A.1: Model of Wireless Sensor Network

path. The distance between two neighboring nodes is a sum up of the physical distance and energy distance. Physical distance depends on the geographical distance of two nodes, and it is a fixed value between two neighboring nodes, while energy distance is relevant to power of battery. Therefore, if the battery of one node is lower than a particular level, the energy distance between itself and its neighboring nodes will increase. So if the battery of one node is charged above or discharged below that level, it has to change its own routing table and broadcast to its neighboring nodes.

A.2 CSP description

We will give an introduction to all of variables, constants, events, channels and processes before we show details about CSP network of WSN.

A.2.1 Variable and Constant

Constant	Type	Meaning
C	Integer	an upper bound of the capacity of Node's battery
L	Integer	a particular level to control the change of routing table
N	Integer	the total number of nodes

Table A.1: Table of constants in WSN

Varible	Type	Meaning
$C(i)$	Float	the current amount of battery of Node(i)
$S(i)$	Set of integers	a set of identical numbers of current available neighboring nodes of Node(i)
$T(i)$	routeTable type	the current routing table of Node(i)
$I(A)$	Integer	the identical number of the neighboring node on the shortest path to the sink
data	any valid types	the routing data inside WSN

Table A.2: Table of varibles in WSN

Note: i is an identical number of a Node, $1 \leq i \leq N$; routeTable is the type of routing table.

A.2.2 Events

Event	Meaning
absorbing	solar panel is absorbing sunshine
charging	charge the battery
processing	the device is doing its work
discharging	discharge the battery
$updating(T(i))$	update the routing table and get a new $T(i)$
$getNodeSets(S(i))$	get the set $S(i)$ of available neighboring nodes
$get(I(A))$	get the identical number $I(A)$ of a neighboring node by algorithm A
sink_processing	the sink is doing further processing

Table A.3: Table of events in WSN

A.2.3 Processes

There are eight processes in a single $Node(i)$, and one process for sink. Processes in $Node(i)$ will be identical with number i, details are shown in Table A.4.

Processor	Meaning
$Solar(i)$	Absorb sunshine when it is allowed
$Charge(i)$	charge the battery when it's possible
$Processor(i)$	get the data from neighboring nodes; control the update of routing table and data forwarding
$Discharge(i)$	discharge the battery when it gets a request
$Device(i)$	a device in Node(i) which needs to consume battery to work
$Update(i)$	update the routing table and find the current $S(i)$ to broadcast
$Forward(i)$	prepare for sending data, and get a neighboring node number $I(A)$
$Output(i)$	Output data to neighboring nodes or sink
$Sink$	Collect data from other nodes

Table A.4: Table of processes in WSN

A.2.4 Channels

A picture of process $Node(i)$ is shown in Figure A.2. The Figure illustrates communication channels between internal processes in $Node(i)$ and two communication channels which are connected with its neighboring node $Node(n)$.

Details about every channel involved in $Node(i)$ and messages through each channel will be elaborated below.

- sun(i): from outside to Solar(i)
  {sunshine} - when the solar panel detects sunshine
  {shadow} - when no sunshine could be detected

- request(i): from Solar(i) to Charge(i)
  {charge} - send a charging request battery
  {stop} - send a command to stop charging battery

- reply(i): from Charge(i) to Solar(i)
  {full} - reply that battery is full
  {start} - send a command to start to charge battery

- battery(i): from outside to Charge(i)
  {full} - send a signal to declare that battery is full

- up1(i): from Charge(i) to Processor(i)
  {update} - when battery reaches a particular level (L) after charging, while battery is lower than L before charging, send this message to update the routing table

- up2(i): from Discharge(i) to Processor(i)
  {update} - when battery is below a particular level (L) after discharging, while battery is higher than L before discharging, send this message to update the routing table
  {start} - start to discharge the battery
  {failed} - when a discharing request is refused

- consume(i): from Processor(i) to Discharge(i)
  {discharge} - when the processor needs to work, send a discharging request

- d_consume(i) from Device(i) to Discharge(i)
  {discharge} - when the device needs to work, send a discharging request
  {fin} - after finishing its work, send a feedback to declare the work has finished

- d_reply(i): from Discharge(i) to Device(i)
  {start} - start to discharge the battery
  {failed} - when a discharing request is refused

- broadcast(i): from Update(i) to Output(i)
  {routeTable, nodeSet} - routeTable is the updated routing table $T(i)$; nodeSet is the set of available neighboring nodes $S(i)$

- route(i): from Forward(i) to Output(i)
  {data, n} - data is what should be sent to the sink; n is an identical number of this neighboring node or the sink

- back(i): from Output(i) to Discharge(i)
  {fin} - after finishing routing data or broadcasting routing table, send a feedback to declare that the data transmission has finished

- i.comin(n): from Output(i) of Node(i) to Processor(n) of Node(n)
  {data} - what should be sent to the sink or routing table of Node(i)

- n.comin(i): from Output(n) of Node(n) to Processor(i) of Node(i)
  {data} - what should be sent to the sink or routing table of Node(n)

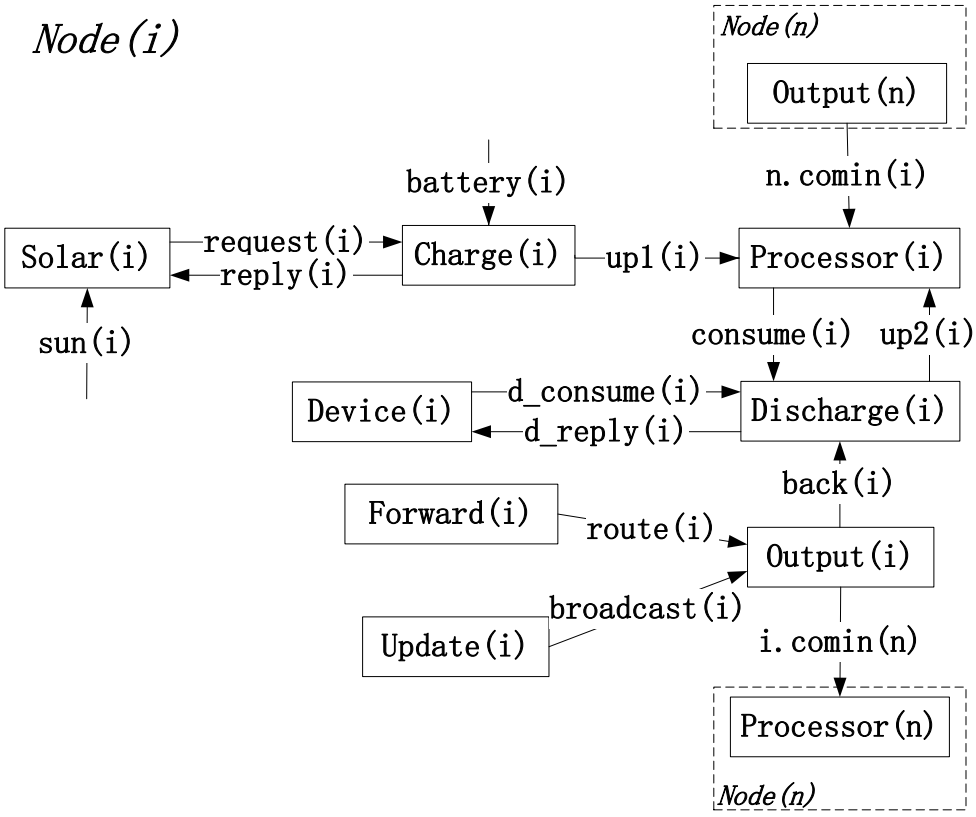


Figure A.2: A picture for $Node(i)$

Process Sink also has its own identity I_{sink}, and it contains channels which are connected with its neighboring nodes. If the sink has one neighboring node $Node(n)$, the picture of Sink is illustrated in Figure A.3.

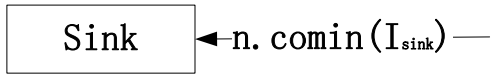


Figure A.3: A picture for Sink

Channel $n.comin(I_{sink})$ is from Output(n) of Node(n) to Sink, with any valid type of message {data} by definition. Sink is the terminal of the data transmission, so after getting the data, it could do further processing.

A.2.5 CSP network

The entire CSP netwrok for WSN is illustrated below:

$$WSN = \|_{i \in N}\ Node(i) \parallel Sink$$

$$Node(i) = Solar(i) \parallel Charge(i) \parallel Processor(i) \parallel Discharge(i) \parallel Device(i)$$

$$\parallel Update(i) \parallel Forward(i) \parallel Output(i)$$

$$Solar(i) = sun(i)?\{sunshine\} \rightarrow request(i)!\{charge\} \rightarrow$$

$$(reply(i)?\{full\} \rightarrow Solar(i))$$

$$\square(reply(i)\{start\} \rightarrow absorbing \rightarrow (reply(i)?\{full\} \rightarrow Solar(i))$$

$$\square(sun(i)?\{shadow\} \rightarrow request(i)!\{stop\} \rightarrow$$

$$Solar(i)))$$

$$Charge(i) = request(i)?\{charge\} \rightarrow$$
$$if\ C(i) == C\ then\ reply(i)!\{full\} \rightarrow Charge(i)$$
$$else\ if\ L \leq C(i) < C\ then\ reply(i)!\{start\} \rightarrow charging \rightarrow$$
$$(battery(i)?\{full\} \rightarrow reply(i)!\{full\} \rightarrow Charge(i))$$
$$\Box(request(i)?\{stop\} \rightarrow Charge(i))$$
$$else\ reply(i)!\{start\} \rightarrow charging \rightarrow$$
$$(battery(i)?\{full\} \rightarrow reply(i)!\{full\} \rightarrow up1(i)!\{update\}$$
$$\rightarrow Charge(i))$$
$$\Box(request(i)?\{stop\} \rightarrow$$
$$if\ C(i) \geq L\ then\ up1(i)!\{update\} \rightarrow Charge(i)$$
$$else\ Charge(i))$$

$$Device(i) = d\_consume(i)!\{discharge\} \rightarrow (d\_reply(i)?\{failed\} \rightarrow Device(i))$$
$$\Box(d\_reply(i)?\{start\} \rightarrow processing \rightarrow$$
$$d\_consume(i)!\{fin\} \rightarrow Device(i))$$

$$Update(i) = updating(T(i)) \rightarrow getNodeSets(S(i)) \rightarrow broadcast(i)!\{T(i), S(i)\}$$
$$\rightarrow Processor(i)$$

$$Forward(i) = get(I(A)) \rightarrow route(i)!\{data, I(A)\} \rightarrow Processor(i)$$

$$
\begin{aligned}
Processor(i) = (up1(i)?\{update\} \rightarrow consume(i)!\{discharge\} \rightarrow \\
(up2(i)?\{start\} \rightarrow Update(i)) \\
\Box(up2(i)?\{failed\} \rightarrow Processor(i))) \\
\Box(up2(i)?\{update\} \rightarrow consume(i)!\{discharge\} \rightarrow \\
(up2(i)?\{start\} \rightarrow Update(i)) \\
\Box(up2(i)?\{failed\} \rightarrow Processor(i))) \\
\Box n.comin(i)?\{data\} \rightarrow \\
if\ data == routeTable\ then\ consume(i)!\{discharge\} \rightarrow \\
(up2(i)?\{start\} \rightarrow Update(i)) \\
\Box(up2(i)?\{failed\} \rightarrow Processor(i)) \\
else\ consume(i)!\{discharge\} \rightarrow \\
(up2(i)?\{start\} \rightarrow Forward(i)) \\
\Box(up2(i)?\{failed\} \rightarrow Processor(i))
\end{aligned}
$$

$$Discharge(i) = (d\_consume(i)?\{discharge\} \rightarrow$$

$$if\ batteryLow\ then\ d\_reply(i)!\{failed\} \rightarrow Discharge(i)$$

$$else\ d\_reply(i)!\{start\} \rightarrow$$

$$if\ C(i) \geq L\ then\ discharging \rightarrow d\_consume(i)?\{fin\} \rightarrow$$

$$if\ C(i) < L\ then\ up2(i)!\{update\} \rightarrow Discharge(i)$$

$$else\ Discharge(i)$$

$$else\ discharging \rightarrow d\_consume(i)?\{fin\} \rightarrow Discharge(i))$$

$$\Box(consume(i)?\{discharge\} \rightarrow$$

$$if\ batteryLow\ then\ up2(i)!\{failed\} \rightarrow Discharge(i)$$

$$else\ up2(i)!\{start\} \rightarrow$$

$$if\ C(i) \geq L\ then\ discharging \rightarrow back(i)?\{fin\} \rightarrow$$

$$if\ C(i) < L\ then\ up2(i)!\{update\} \rightarrow Discharge(i)$$

$$else\ Discharge(i)$$

$$else\ discharging \rightarrow back(i)?\{fin\} \rightarrow Discharge(i))$$

$$Output(i) = (broadcast(i)?\{T(i), S(i)\} \rightarrow \|_{j \in S(i)}\ i.comin(j)!\{T(i)\} \rightarrow back(i)!\{fin\}$$

$$\rightarrow Output(i))$$

$$\Box(route(i)?\{data, n\} \rightarrow i.comin(n)!\{data\} \rightarrow back(i)!\{fin\} \rightarrow Output(i))$$

$$Sink = n.comin(I(sink))?\{data\} \rightarrow sink\_processing \rightarrow Sink$$

Bibliography

[1] The Theory and Practice of Concurrency, Roscoe, A. W., Prentice Hall, ISBN 0-13-674409-5, 1997

[2] A Brief History of Process Algebra, J.C.M. Baeten, Technische Universiteit Eindhoven, 2004

[3] Communicating Sequential Processes, C.A.R. Hoare, 2004-06-21

[4] A Calculus of Communicating Systems, Robin Milner, Springer Verlag, ISBN 0-387-10235-3,1980

[5] Algebra of Communicating Processes, J.A. Bergstra, J.W. Klop, 1985

[6] http://en.wikipedia.org/wiki/C._A._R._Hoare

[7] ForSyDe tutorial, Alfonso Acosta, 2008-08-15

[8] A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors, Olivier Beaumont, Vincent Boudet and Yves Robert, 2002

[9] Synchronous Data Flow, Edward A. Lee, David G. Messerschmitt, 1987

[10] DEHAR: a Distributed Energy Harvesting Aware Routing Algorithm for Ad-hoc Multi-hop Wireless Sensor Networks, Mikkel Koefoed Jakobsen, Jan Madsen, and Michael R. Hansen, 2010