Y-chart Application Programmer's Interface

The YAPI Programmer's and Reference Guide

Version 2.1.0

Erwin de Kock and Gerben Essink

Contents

Preface

Aim

YAPI is an acronym for Y-chart Application Programmer's Interface. The goal of YAPI is to enable reuse of stream processing applications in two ways namely composability and portability. Composability refers to the use of existing stream processing applications in the development of new stream processing applications. Portability refers to the use of stream processing applications on different architectures. To this end, YAPI separates the concerns of the application designer, who specifies the functionality of the system, and the system designer, who implements the functionality.

Intended Audience

This document is primarily intended for designers of signal processing applications serving as a programmer's manual. Although YAPI itself is programmed using the C++ language, application programmers require only some basic knowledge of C++ in order to program the interfaces of the processes and the structures of the process networks. Knowledge of C suffices to describe the functionalities of the processes. For an introduction to C++ we refer to [19]. We provide an efficient implementation of YAPI in the form of a C++ run-time library to execute the applications on a workstation. Subsequently, the applications are used by the system designer as input for mapping and performance analysis in the design of complex stream processing systems.

It is our aim to offer YAPI to colleagues in Philips with expertise in stream processing applications such that they can model their applications in a structured way. These applications can serve as input for our methodologies and tools. YAPI has been been applied in several research projects among others COdesign Simulation and sYnthesis (COSY), PROgrammable MultiMedia PlaTforms II (PROMMPT-

II), SCalable BAseband architecture for 3-G terminals (Scuba), Space Computer Architecture 4 Killer Experience (SpaceCAKE), Video Chain Optimization (VCO), Scalable Video Algorithms (SVA), New Algorithms for Video Enhancement, and MPEG coding and Processing for Storage (MEPS). Outside research YAPI is being applied among others at the System Lab Eindhoven in the projects Function and Architecture modeling Kit and Interface Rules (FAKIR), and Universal Display Processor (UDP).

Acknowledgements

The development of YAPI has been initiated by the members of the former hardware software codesign cluster of the group Niessen in cooperation with the former members of the processor oriented architectures cluster of the group Lippmann working in the COSY project. The authors gratefully acknowledge the contributions to YAPI from Jean-Yves Brunel, Arjan Kenter, Wido Kruijtzer, Paul Lieverse, Wim Smits, Kees Vissers, and Pieter van der Wolf.

YAPI Release 0.5 has been developed in the cluster Embedded Systems Design Technology of the group Embedded Systems Architectures on Silicon. The YAPI development is part of an effort to develop methodologies and tools to support the design of stream processing systems. Currently, YAPI is being developed further in cooperation with Eric Kamps and Martin Klompstra from Electronic Design and Tools Synthesis who will distribute this release as YAPI ED&T Release 1.0 and any further releases.

Outline

The outline of the document is as follows.

Chapter 1 describes the basic concepts of YAPI and the typical design flow in which it is applied. Furthermore it describes the model of computation and related work in this area. Since this chapter introduces the terminology for the rest of this document we suggest that you read this chapter.

Chapter 2 is the application designer's guide. This chapter should be read by application designers who wish to write or understand YAPI applications. The outline of this chapter is as follows.

Section 2.1 is a tutorial in which the programming primitives are introduced

step by step by following the design of a simple application. This section is intended for newbies in YAPI application programming.

Section 2.2 describes all the details you need to know when you write complex application programs with YAPI. This section is intended for the more experienced users.

Section 2.3 describes a run-time environment that you can use to execute YAPI applications.

Section 2.4 explains how you can print computation and communication statistics that have been measured during the execution of an application in the run-time environment.

Section 2.5 describes how you can visualize the structure of process networks.

Section 2.6 describes programming guidelines containing the preferred coding style and some design rules. The coding style aims at a uniform layout for all application programs such that they are easier to understand and maintain. The design rules aim to improve the reusability of application programs.

Section 2.7 is the YAPI reference for application designers. It describes the public interface of the YAPI C++ classes that can be used in YAPI applications.

Section 2.8 explains how you can compile your applications.

Chapter **??** describes the installation procedure for the YAPI run-time library.

Chapter 1

Introduction

1.1 System Level Design

Modern stream processing systems such as digital televisions, set-top boxes, and mobile devices are multi-functional systems that support multiple standards. For example, consider a digital TV set which receives an MPEG transport stream via digital video broadcast. In this TV set the MPEG transport stream is demultiplexed into a number of MPEG video streams, and these MPEG video streams are decoded, scaled, and merged for display. Via a remote control unit a user can select the channels to be displayed, including the size and position of the video windows.

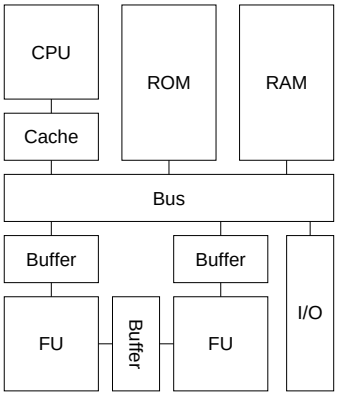

Figure 1.1: Embedded System Architecture

Now assume that a system designer is facing the challenge to implement the functionality in an embedded system. In embedded systems the user-interface functionality is typically handled in software because the system should have the flexibility to deal with rapidly changing functional specifications. For other functions a software implementation is essential to be able to deal with multiple standards or with multiple languages in different countries, for example. The embedded system shown in Figure 1.1 has one CPU on which the software tasks run, for instance, to handle the events from the remote control. For the video processing functions like MPEG decoding and video scaling, a software implementation may be inappropriate because of the high data rates that are involved. The performance requirements and the constraints on silicon area and power consumption require that significant parts of the system are implemented in dedicated hardware blocks that run in parallel with the CPU. As a consequence, the system has a heterogeneous architecture, i.e., it consists of programmable and dedicated components. Besides the processors the system also contains other elements. A ROM may contain program code for the software tasks that run on the CPU. RAM, caches and buffers provide storage for temporary data, and a bus provides resources for inter-processor communication.

In our view the design of these heterogeneous multi-processor architectures should follow a general scheme, visualized by the Y-shape in Figure 1.2, hence, the name

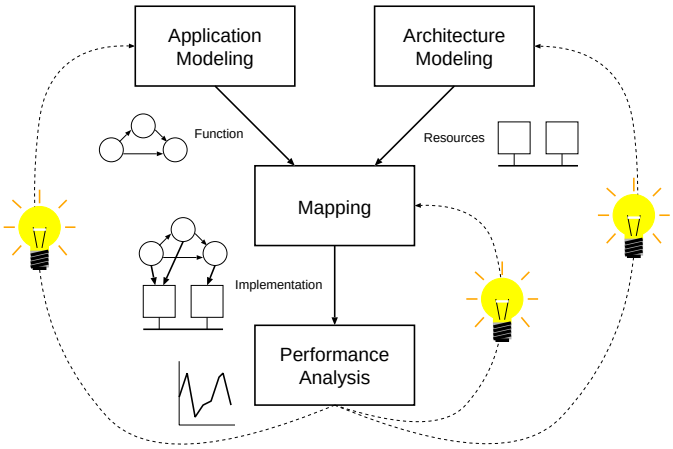

Figure 1.2: The Y-chart: a general scheme for heterogeneous system design.

Y-chart [18] [1].

In the Y-chart we distinguish between four activities. The first activity is Application Modeling. The purpose of this activity is to capture a functional specification of the system in the form of a set of benchmark applications. The application modeling is done by a person we refer to as application designer. The second activity in the Y-chart is Architecture Modeling. The purpose of this activity is to model the resources that are available in the system. In embedded systems these resources typically are processors, operating systems, buses and memories. Another important resource is time. In the third activity, the Mapping, the function is mapped onto the resources of the architecture. The result of the mapping is an implementation of the system which can be used as input for the fouth activity, Performance Analysis.

Typically, a system designer studies the set of benchmark applications, makes some initial calculations and proposes an architecture. Architectures are evaluated quantitatively by means of performance analysis. To this end, each application is mapped onto the architecture and the performance of each application-architecture-mapping combination is evaluated. The resulting performance numbers may inspire the architecture designer to improve the implementation via another mapping, or by choosing other resources in the architecture. He may also decide to restructure the applications, for instance if the performance requirements cannot be met with the given resources. The system designer actions are denoted by the light bulbs in Figure 1.2.

1.2 Application Modeling

With YAPI we focus on design technology for capturing the functional specifications of signal processing applications. The design technology for capturing the functional specifications of signal processing applications must satisfy a diverse set of requirements. The design technology must provide support for

- composing applications into larger applications to enable modular construction and reuse,
- importing legacy code that is available in the form of C code,
- executing applications to validate the functionality of the system by processing real data sets,
- measuring computation and communication requirements related to the processing of a data set to determine the workload imposed on target architectures, and
- explicit expression of communication and parallelism to enable the mapping of applications onto target architectures in order to provide a link to architecture design technologies.

To meet the above-mentioned requirements we developed YAPI as a C++ library with a set of rules which can be used to model stream processing applications as a process network. The model of computation of YAPI is based on Kahn Process Networks. Such a network consists of concurrent processes that communicate via unbounded fifos. In Kahn Process Networks (KPN) parallelism and communication are explicitly modeled, which is essential for the mapping onto multi-processor systems. Another property of KPN is that an application designer can combine processes into networks without specifying their order of execution. This property stimulates the modular construction and reuse of applications (functional IP), since it is easy to compose new applications using existing ones.

An example process network is shown in Figure 1.3. This process network represents the stream processing functionality of the digital TV set introduced in Section 1.1. A process network consists of concurrent processes that are interconnected by directed first-in-first-out channels which are called fifos. In Figure 1.3 the processes are represented by circles. Each process has a private state space which is not accessible by other processes. The processes have input and output ports through which they communicate with their environment. Each port is connected to precisely one fifo, and each fifo is connected to exactly one output port and exactly one input port. In the figure the ports are represented by black dots, and the fifos are represented by arrows. The structure of a process network, i.e., the

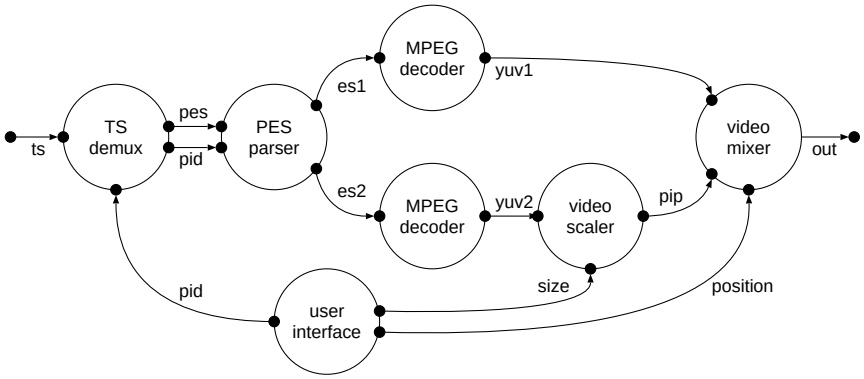

Figure 1.3: A Process Network

binding of fifos to ports, is static; it is known at compile time and does not change at run time. The ports and the fifos are typed which means each instance can handle data of one type only. Different instances can handle different data types.

A process performs a sequence of computation and communication actions that may be arbitrarily interleaved. With YAPI the computations can be specified by any C or C++ construct, whereas the following primitives can be used for communication.

A read action removes data from the fifo and stores it into a local variable of the process, if data is available. If there is no data the read action will block. Depending on the implementation the read may incur context switching with associated state saving. In YAPI, the state saving is implicit since it is invisible to the application designer. The write function copies data from a local variable into the fifo. In theory the fifos are unbounded, so a write action will never block. In practical implementations such as in embedded systems where the amount of memory is limited, the size of a fifo is bounded. Therefore the write action will block if the fifo is full to avoid data loss during communication.

Note that the above-mentioned primitives ensure deterministic behavior, i.e., the same input always results in the same output, and that the size of the fifos determines the set of feasible process schedules. If the fifos are unbounded, then we have maximal process parallelism and we obtain the model of computation known as Kahn Process Networks [16]. If the size of the fifos is zero, then we have minimal process parallelism and we obtain a subset of the model of computation known as Communicating Sequential Processes [17].

In some cases, for instance, in reactive systems where the communication behavior of the environment is unknown, the deterministic communication provided by read and write may be insufficient. To be able to deal with reactive systems we extended

the model with non-deterministic communication via the select action. A select action will select one of the ports that will eventually produce or consume the given amount of data. In the example below the select is used to handle the input of a new window width from the window manager.

1.3 Example

We illustrate the use of YAPI with an example of a filter that scales video lines. The purpose of the example is twofold. First we illustrate the combination of deterministic and non-deterministic communication. Second we illustrate the decoupling of the data types that are used for communication and computation. To this end we assume that the filter receives a stream of window widths from a window manager through input port p and a stream of pixels from a video source through input port q. The function of the filter is to scale the incoming video frames according to the incoming window widths and to transmit the resulting stream of pixels through output port r as outlined in the code fragment shown in Figure 1.4.

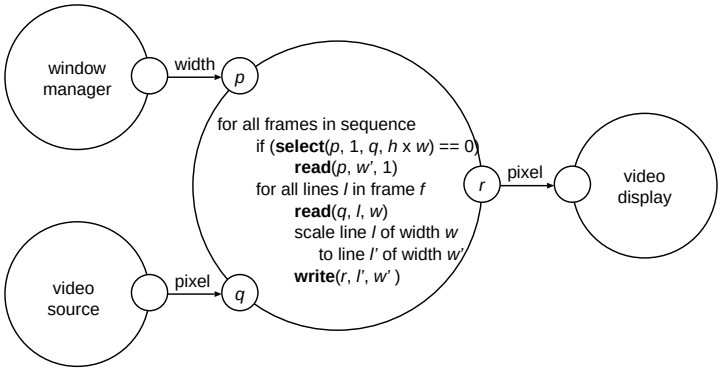

Figure 1.4: Horizontal Video Scaling

The video source and the window manager are not synchronized. We assume that the video source produces frames at a constant rate using write action *write*$(p', f, h \times w)$, where p' is an output port and f is a video frame of h video lines that each contain w video pixels. The behavior of the window manager is unknown because it is controlled by the user. In order to cope with the non-deterministic behavior of the window manager we guard the read action of the window width in the filter with the select action *select*$(p, 1, q, h \times w)$. If the select action returns 0 then this indicates that port p can deliver one token. Hence, we initiate the read action *read*$(p, w', 1)$ to obtain the new window width w' without the danger of blocking forever. If the select action returns 1 then this means that port q can deliver $h \times w$ tokens. Hence, we initiate the filtering of the next video frame. The resulting latency between input and output depends on the granularity of the filter. The filter shown in the example

is line-based which means that it reads the video frame by h consecutive read actions *read*(q, l, w) where l represents a video line of w pixels thereby introducing a latency of one video line. Frame-based and pixel-based filters are also feasible. Note that in this scenario it is possible to scale two or more video frames to the same window width, but that it is not possible to change the width of a window in the middle of a video frame. In order to specify the latter, for instance to allow non-rectangular windows, the select action should be moved to the inner loop that iterates over the video lines.

1.4 Related Work

Kahn process networks [16] is a model of computation that is often used for modeling stream processing applications. In this model, concurrent processes communicate through unidirectional first-in-first-out channels with unbounded capacity. Each of the processes performs sequential computation on its private state space. The computation actions are interleaved with communication actions that read data from input channels and write data to output channels. Read actions are blocking, i.e., a process that reads from an empty channel stalls until the channel has sufficient data to complete the read action. Write actions are non-blocking because the channels have unbounded capacity. A well-known property of a Kahn process network is that it is deterministic, i.e., the stream of data that travels along each channel is determined by the given input data; it does not depend on the order in which the processes are executed. For this reason, an application designer can combine processes that represent stream processing functions into process networks without specifying their order of execution. Moreover, a system designer can exploit the concurrency between the processes by using processing elements that operate in parallel.

In some implementations of Kahn process networks, the blocking read semantic incurs a considerable amount of context switching overhead. Dataflow process networks [15], which are a special case of Kahn process networks, avoid this overhead by transforming the processes into atomic actors that are fired when input data is available. Once an actor has been fired, it cannot stall. If it cannot complete the computation because it requires more input data, then it must save its internal state such that it can resume the computation on the next firing. The literature contains many references to variants of dataflow process networks such as synchronous or static dataflow [7], cyclo-static dataflow [14], and dynamic dataflow [2]. Many commercial vendors offer software packages for modeling signal processing systems based on dataflow process networks. Examples of such packages are SPW [11] and DSP Station [5]. Another example of a software package that is based on the data-flow paradigm is SLAM+C [21] which was developed at Philips Research.

We argue that dataflow process networks are less suited for modeling data-dependent applications because they place the burden of state saving on the application designer rather than on the system designer. Explicit implementation of state saving in a dataflow program is a form of over-specification which can lead to unnecessary computation or communication. The reason for this is that the need for state saving and the implementation of state saving are design decisions. A system designer can avoid the need for state saving by avoiding resource sharing. If a system designer decides to apply resource sharing then an on-line resource scheduler with multi-tasking capabilities can provide automatic state saving. Only in case of resource sharing without multi-tasking capabilities there is a need for explicit state saving. For this reason we resort to the more general model of Kahn process networks which assumes implicit state saving, thereby leaving the use and the implementation of state saving as design decisions to a system designer.

A limitation of Kahn process networks is that they cannot model reactiveness such as user interaction. This is caused by the fact that the absence and the occurrence of non-deterministic events cannot be made known to the processes. Control flow models such as finite state machines provide a solution for this problem by assuming a broadcast mechanism to communicate events in which each actor is sensitive to specific events. These models often contain a global notion of time such that time stamps can be associated with all events which is needed to process them in a correct order. This makes these models less suited for the implementation of computationally intensive applications because the amount of parallelism is limited. Furthermore, the underlying broadcast mechanism of these models is difficult to implement on parallel systems with distributed memory. Examples of software packages supporting control flow modeling are Statecharts [12], Esterel [6], and Polis [1].

The Ptolemy system [10] has been designed to support a heterogeneous mix of models of computation for co-simulation. It attempts to combine the semantics of control and data flow models at their interfaces [3]. Although this is feasible for functional simulation, we argue that this does not allow hardware software co-design because the models of computation are already tuned towards a target implementation. Another approach called Process Coordination Calculus [4] combines data-driven and event-driven processes in a single process network with stream-based, event-based, and register-based communication schemes. A specification consists of a process network and a set of scheduling constraints to ensure deterministic behavior.

To extend the deterministic model of Kahn process networks with non-deterministic events we pursue the approach of Martin [9], who has introduced a communication primitive known as the probe in combination with the model of Communicating Sequential Processes [17]. In this model concurrent processes communicate through unbuffered channels. As a result two communicating processes must complete

their communication actions simultaneously. A probe action indicates whether the process on the opposite side of the channel is stalling because it has initiated a communication action that cannot be completed. Martin [13] demonstrates the use of probes to implement channel selection, i.e., selection of one channel out of a set of channels such that the next communication action on the selected channel can be completed. Since channel selection is performed at run-time it allows the modeling of non-deterministic events.

We generalize the notion of probes to buffered channels in order to extend the model of Kahn process networks with channel selection. Although channel selection can be implemented with probe actions, YAPI provides a more abstract operation because we argue that the implementation of channel selection is the concern of a system designer rather than of an application designer. We hide the algorithm that selects a channel when there is more than one candidate from an application designer, because the conditions that determine whether or not a channel is a candidate depend on design decisions such as the scheduling of shared resources, the computation delays, the communication delays, and many more. Since an application designer does not know these design decisions, we do not allow an application designer to control the channel selection algorithm. Therefore, we avoid probe-like constructs in YAPI that allow application designers to implement their own selection algorithm. We claim that the abstraction from these implementation details results in reusable applications that can be mapped onto different target architectures.

1.5 Model of Computation

To describe the structure of a process network, we introduce the notions of process type set and data type. A process type set defines the set of process types. Each process type has a set of input ports and a set of output ports. With each port we associate a data type. A process network consists of a set of processes and a set of channels. A channel connects a process output port to a process input port of the same data type. Each port is connected to precisely one channel.

To describe the communication between the processes we provide three functions called *read*, *write*, and *select* that can be called from within a process. The informal meaning of these functions is as follows. The read function consumes data from an input port and stores it in a local variable of the process. The write function copies the value of a local variable to an output port. The select function selects an input or output port that eventually will produce or consume data, respectively. To formalize the semantics of these functions we associate with each port at any time the number of tokens that it has transferred up to now and the number of tokens that it has committed to transfer up to now but that not have been transferred yet.

To this end we introduce the following definitions.

Definition 1.5.1 Let p be a port. Then at any time

- $c(p)$ denotes the number of tokens transferred through p,
- $n(p)$ denotes the number of tokens committed but not yet transferred through p,
- $m(p)$ denotes the port connected to p by a channel, and
- $v(p, k)$ denotes the value of the k-th token at p.

Following the approach of Martin [8], the communication mechanism is based on the following assumptions. For all channels it must hold at any time that a write action is not blocked, the number of consumed tokens does not exceed the number of produced tokens, and the functionality is first-in-first-out.

Axiom 1.5.1 Let $(p, m(p))$ be a channel from output port p to input port $m(p)$. Then the following invariants hold

- $n(p) = 0$,
- $c(m(p)) \leq c(p)$, and
- $\forall_{0 \leq i < c(m(p))} v(m(p), i) = v(p, i)$.

Under these assumptions we define the semantics of the read, write, and select functions using preconditions and postconditions as follows. Note that these functions stall when their postcondition cannot be satisfied. Furthermore, note that the write function is non-destructive which means that the variables of the producing process keep their values.

Definition 1.5.2 Let p be an input port of type t, x an array of type t, and n a positive integer indicating a number of tokens. Then action *read*(p, x, n) is defined by

- precondition $c(p) = N$, and
- postcondition $c(p) = N + n \wedge \forall_{0 \leq i < n}(x[i] = v(p, N + i))$.

Definition 1.5.3 Let p be an output port of type t, x an array of type t, and n a positive integer indicating a number of tokens. Then action *write*(p, x, n) is defined by

- precondition $c(p) = N \wedge \forall_{0 \leq i < n}(x[i] = v(p, N + i))$, and
- postcondition $c(p) = N + n \wedge \forall_{0 \leq i < n}(x[i] = v(p, N + i))$.

Definition 1.5.4 Let k be a positive integer, p_1 up to p_k ports, n_1 up to n_k positive integers indicating requests for numbers of tokens, and s a positive integer no larger than k indicating the index of the selected port. Then action $s =$*select*$(p_1, n_1, \ldots, p_k, n_k)$ is defined by

- precondition $\forall_{1 \leq i \leq k}(c(p_i) = N_i)$, and
- postcondition $\forall_{1 \leq i \leq k}(c(p_i) = N_i) \wedge 1 \leq s \leq k \wedge \Diamond(N_s + n_s \leq c(m(p_s)))$.

Following temporal logic theory, the symbol $\Diamond$ means *eventually*. Hence, the select function selects a port such that the current number of transferred tokens through this port plus the requested number of tokens is eventually smaller than or equal to the number of transferred tokens through the connected port. If we consider an input port p_s, then this port is a candidate for selection if the corresponding output port $m(p_s)$ will eventually produce enough tokens to complete a read action of n_s tokens. If we consider an output port p_s, then this port is a candidate for selection if the corresponding input port $m(p_s)$ will eventually consume the tokens produced by a write action of n_s tokens. Note that the select function has no effect on the number of transferred tokens, i.e., it does not produce or consume data.

To describe the functionality of the processes we use a sequential programming language. We introduce an additional function called *execute* to abstract from the implementation of the functionality in the sequential programming language. To this end, the functionality between two communication actions has to be annotated with one or more execute actions. These execute actions annotate the computation requirements of the processes; they do not provide additional functionality.

1.6 Mapping

The implementation of read, write, select, and execute actions are a concern of a system designer. Note that different read, write, select, and execute actions may be implemented in different ways, for instance, because some actions are executed in hardware and other actions are executed in software.

One of the design decisions is to determine the size of the fifos in order to obtain an implementation in finite memory. Deadlock can occur if they are too small, because the size of the fifos limits the set of reachable schedules. Going from unbounded to bounded fifos changes Axiom 1.5.1 such that for all channels it holds that at any time a write action and a read action are not blocked simultaneously, the number of produced tokens minus the number of consumed tokens is bounded by the size of the channel, and the functionality is first-in-first-out. Formally, this is denoted as follows.

> **Axiom 1.6.1** Let $(p, m(p))$ be a channel of size s. Then the following invariants hold
>
> - $(n(p) = 0) \vee (n(m(p)) = 0)$,
> - $0 \leq c(p) - c(m(p)) \leq s$, and
> - $\forall_{0 \leq i < c(m(p))}(v(m(p), i) = v(p, i))$.

The read and write actions can be implemented such that the number of communicated tokens can exceed the size of the fifo. To this end, these actions have to be preempted when the number of committed tokens is not present or does not fit in the fifo.

Another design decision is the implementation of the notion of 'eventually' in the select function. For process networks that cannot be scheduled off-line, for instance due to data-dependent functionality, we have chosen to strengthen the expression $\Diamond(N_s + n_s \leq c(m(p_s)))$ in the postcondition of the select function to $c(p_s) + n_s \leq c(m(p_s)) + n(m(p_s))$. We obtain the number of committed tokens $n(m(p_s))$ through the number of tokens of the read and write actions, i.e., the initiation of a read or write action of n tokens sets a commitment that decreases during transfer of the tokens. As a result the scheduling horizon is limited to one communication action, i.e., a select action takes one incomplete read or write action into account. Again this introduces deadlock if the size of the fifos is too small. Note that if the process network is a dataflow process network, then the firing rules can be implemented with select actions. In that case the read and write actions do not have to stall because the firing rules satisfy Axiom 1.6.1.

The above-mentioned design decisions are used in the YAPI run-time library that is used by application designers to simulate the functionality of a process network on a workstation. Other YAPI implementations, in particular those proposed in COSY [22] and SPADE [20], target mixed hardware and software realizations in systems-on-chip. In the initial stage of the design process, these implementations are abstract performance models to allow fast design space exploration. Subsequently, these performance models are refined into cycle-accurate models to allow final implementation.

Chapter 2

Application Programmer's Guide

2.1 Getting Started

2.1.1 Introduction

To get into YAPI as quickly as possible, we illustrate the concepts by following the design of a Producer-Consumer application, which is the *Hello World* type of application for YAPI. The example is fully compliant with the coding style as presented in Section 2.6.1, but to avoid that you immediately get lost in details only the very essential features of YAPI are used. The more advanced features of YAPI are introduced in Section 2.2, and a complete reference can be found in Section 2.7. The complete source code of the Producer-Consumer example can be found in the YAPI Applications package.

We assume that you are already familiar with C++ concepts, such as classes, constructors, (virtual) member functions, inheritance, function overloading, templates, and references. For a good introduction to C++ we refer to [19].

2.1.2 Producer-Consumer

In the Producer-Consumer example we have two processes, a *producer* which writes integer values to a channel, and a *consumer* which reads the values from the channel. The first value that is written by the producer indicates the number of values that will follow.

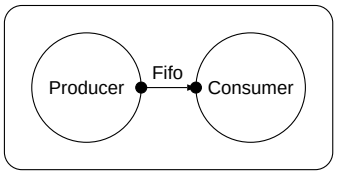

Figure 2.1: Producer-Consumer

In the remainder of this section we will discuss the Producer-Consumer example in a top-down way. We start with the main program in which the process network is created and started. Next we describe the process network, followed by the producer process and the consumer process.

2.1.3 The Main Program

To run a YAPI application one must first create a process network, and next call the YAPI function `start` to execute it. In its simplest form this is done in the

`main` function as shown in Program 2.1.1. Assume that the class `PC` represents the Producer-Consumer process network which is declared in the file `pc.h`. We create the network with the declaration `PC pc(id("pc"))`. This may look complicated, but basically what happens is that we create a variable `pc` which is a process network of type `PC` with identification string `"pc"`. Next we execute the network using the `start` function from the YAPI Run-Time Environment. The `start` function is declared in the include file `yapi.h`.

Program 2.1.1 The Main Program (main.cc)

```
#include "pc.h"
#include "yapi.h"

int main()
{
        RTE rte;

        // create the process network
        PC pc( id("pc") );

        // start the process network
        rte.start(pc);

        return 0;
}
```

2.1.4 The Process Network

The Producer-Consumer process network is represented by the class `PC`. The declaration of this class can be found in the file `pc.h`, and the implementation can be found in the file `pc.cc`.

Program 2.1.2 Process Network Declaration (pc.h)

```
#include "yapi.h"

#include "producer.h"
#include "consumer.h"

class PC : public ProcessNetwork
{
public:
        PC(const Id& n);
        const char* type() const;

private:
        Fifo<int> fifo;

        Producer producer;
        Consumer consumer;
};
```

The class `PC` is derived from the YAPI class `ProcessNetwork`. The class has two public member functions, namely a constructor and a type function. These two functions must be defined in all process networks. The constructor is used to create the network in the main program, and the type function may be used to identify the type of the network as a string, for instance for debugging. Next, the class has private members, which in case of a network are the fifos and the processes that constitute the network. In case of the Producer-Consumer network we have a fifo that conveys values of type `int`, a process of type `Producer` called `producer`, and a process of type `Consumer` called `consumer`. The fifo is declared in the YAPI include file `yapi.h`, and the processes are declared in the user-defined include files `producer.h` and `consumer.h`, respectively.

Program 2.1.3 Process Network Implementation (pc.cc)

```
#include "pc.h"

PC::PC(const Id& n) :
        ProcessNetwork(n),
        fifo    (id("fifo")),
        producer(id("producer"), fifo),
        consumer(id("consumer"), fifo)
{ }

const char* PC::type() const
{
        return "PC";
}
```

The implementation of the class `PC` can be found in the file `pc.cc`. In this file we find an implementation of the constructor and the type function. The constructor first initializes the base class, `ProcessNetwork`, and next it initializes the private members, which are the fifo and the two processes. The initialization of these objects is performed by calling their constructors. The constructor of a fifo has one argument, namely its identification string. The constructor of a process also has an identification argument, and one or more fifos that are connected to its input and output ports. In this example the fifo is passed as an argument to the constructor of the producer, where it is bound to the output port, and the fifo is also passed as an argument to the constructor of the consumer where it is bound to the input port. The type function of a process network returns the class name as a string, which is `"PC"` in this case.

2.1.5 The Producer

The producer process is represented by the class `Producer`. The class is declared in the include file `producer.h` and its implementation can be found in the file `producer.cc`.

A user-defined process must be defined by a class that is derived from the YAPI class `Process`. It must have a constructor, a type function, and a main function. The purpose of the constructor and the type function are similar to those of a process network. Additionally, a process has a main function which represents the functionality of the process. A process network has no main function because its functionality is represented by its processes and fifos. A process has one or more input or output ports which must be declared as private member variables such that they can be accessed by the process from the main member function but not by other processes. The producer has one output port of type `int`, called `out`. The classes `OutPort` and `Process` are YAPI classes which are declared in include file `yapi.h`.

The constructor of the producer has two arguments, an identification of type `Id`, and an output object of type `Out<int>`. Template class `Out` is a base class of both `Fifo` and `OutPort` such that both fifos and output ports can be passed as arguments. In this example output port `out` is connected to fifo `fifo` in Program 2.1.3. In Section 2.2.1 we will show an example in which an output port is connected to another output port.

Program 2.1.4 Producer Process Declaration (producer.h)

```
#include "yapi.h"

class Producer : public Process
{
public:
        Producer(const Id& n, Out<int>& o);
        const char* type() const;
        void main();

private:
        OutPort<int> out;
};
```

Program 2.1.5 Producer Process Implementation (producer.cc)

```
#include "producer.h"
#include <iostream>

Producer::Producer(const Id& n, Out<int>& o) :
        Process(n),
        out( id("out"), o)
{ }

const char* Producer::type() const
{
        return "Producer";
}

void Producer::main()
{
        std::cout << "Producer started" << std::endl;

        const int n = 1000;

        write(out, n);

        for (int i=0; i<n; i++)
        {
                write(out, i);
        }

        std::cout << type() << " "
                  << fullName() << ": "
                  << n << " values written"
                  << std::endl;
}
```

The implementation of the class `Producer` can be found in the file `producer.cc`. In this file we find an implementation of the constructor, the type function, and the main function. In the constructor we first initialize the base class, `Process`, and next we initialize the private member, which is the output port. The constructor of an output port has two arguments, namely its identification string, and an output object, which could be a fifo, or another output port as we will see later.

The type function of class `Producer` returns the string `"Producer"`. In the main function the constant `n` defines the number of tokens to be transferred. First the number of tokens is written to output port `out`, and next, in the for-loop, the numbers are written. Finally, before exiting the main function the type and the name of the process are printed to indicate that the process has terminated successfully.

2.1.6 The Consumer

The consumer process is similar to the producer. The differences are that it has an input port instead of an output port, and that data is read from this port instead of being written.

Program 2.1.6 Consumer Process Declaration (consumer.h)

```
#include "yapi.h"

class Consumer : public Process
{
public:
        Consumer(const Id& n, In<int>& i);
        const char* type() const;
        void main();

private:
        InPort<int> in;
};
```

Program 2.1.7 Consumer Process Implementation (consumer.cc)

```
#include "consumer.h"
#include <assert.h>
#include <iostream>

Consumer::Consumer(const Id& n, In<int>& i) :
        Process(n),
        in( id("in"), i)
{ }

const char* Consumer::type() const
{
        return "Consumer";
}

void Consumer::main()
{
        int n,j;

        std::cout << "Consumer started" << std::endl;

        read(in, n);

        for (int i=0; i<n; i++)
        {
                read(in, j);
                assert(i==j);
        }

        std::cout << type() << " "
                  << fullName() << ": "
                  << n << " values read"
                  << std::endl;
}
```

The definition of the consumer completes the Producer-Consumer example. If you are new to YAPI, and you wish to experiment with some toy YAPI program, we suggest that you start with the Producer-Consumer example, and add for example an extra process between the producer and the consumer. This process could simply pass the data, or do some “real” processing. You may also create an input process that reads data from a file, or an output process that writes data to a file.

2.2 Carrying On

2.2.1 Hierarchical Process Networks

With YAPI it is possible to create hierarchical process networks. A hierarchical network is a network that contains other process networks. The purpose of such a hierarchy is to raise the level of reuse from processes to process networks. A process network is also an abstraction of some functionality, similar to a process. This abstraction hides information which is essential for managing the complexity.

To be able to use a process network as a component within another network, the process network must have ports to let it communicate with its environment. Here, we present an example of such a process network that can be used as a component within another network. The process network scales pictures in the horizontal and vertical direction. The process network has two ports, `in` and `out`, two subprocesses, `x` and `y`, and an internal fifo channel `f`.

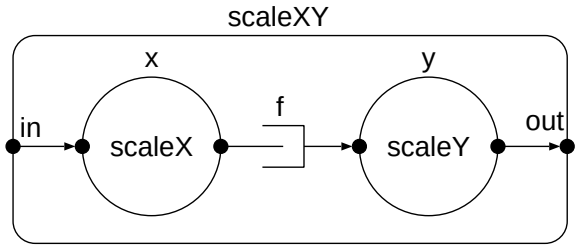

Figure 2.2: Horizontal and Vertical Scaler

Program 2.2.1 ScaleXY Process Network Declaration

```
class ScaleXY : public ProcessNetwork
{
public:
        ScaleXY(const Id& n, In<int>& i, Out<int>& o);
        const char* type() const;

private:
        InPort<int>  in;   // input port
        OutPort<int> out;  // output port
        Fifo<int>    f;    // internal fifo
        ScaleX       x;    // horizontal scaler
        ScaleY       y;    // vertical scaler
};
```

The network structure is created by the constructor, which first initializes the ports,

then the internal fifos and finally the internal processes. In this network, the input port of the horizontal scaler is connected to the process network input port, `in`. The output port of the vertical scaler is connected to the output port of the process network, `out`. The output port of the horizontal scaler and the input port of the vertical scaler are connected to the internal fifo of the process network. As this example shows, the ports of the internal processes (and internal process networks) can be connected to internal fifos, or to the ports of the process network. The ports of the process network will be connected to external fifos, or the network ports of the surrounding network. There is no limit to the nesting of process networks.

Program 2.2.2 ScaleXY Process Network Implementation

```
ScaleXY::ScaleXY(const Id& n, In<int>& i, Out<int>& o) :
        ProcessNetwork(n),
        in (id("in"),  i),
        out(id("out"), o),
        f  (id("f")),
        x  (id("x"), in, f),
        y  (id("y"), f, out)
};
```

It is important to note that the order in which the constructors of the ports, fifos, and processes are called is the same as the order in which they are declared in the header file. Therefore, always declare ports first, fifos next, and processes and process networks last. If you do not to comply to this rule, then the connection between ports and fifos may fail which causes unexpected behavior. Furthermore, note that you do not directly connect the input (output) of process `x` (`y`) to variable `i` (`o`), since you are then bypassing the ports of the process network. Therefore, always connect processes and process networks to member variables (ports and fifos) only.

Identification

As we saw earlier, process networks can be hierarchical. In fact, all YAPI objects, such as processes, ports, fifos, and process networks are part of a network hierarchy. In such a hierarchy every object, except the root process network, has a *parent object*. The parent of a port is a process or a process network. The parent of a fifo, the parent of a process, and the parent of a process network is a process network. The class `Id` has been introduced to identify the objects in the hierarchy, and to be able to retrieve information about these objects at run-time. This information may be used for many purposes, such as debugging.

The class `Id` provides the member functions `parent`, `name`, and `fullName`, to obtain the parent object, the *name*, and the *full name* of the object, respectively. The *name* of an object refers to the instance name of the object, represented by a null-terminated character string. The full name of an object is a representation of the complete path of the object in the network hierarchy. The full name is constructed by concatenating the names of the objects on the path starting at the root network, separated by dots. For example, the full name of output port *out* in process *producer* in process network *pc* is *pc.producer.out*. We recommend that you make the full names of objects unique, however this is not mandatory.

The classes `ProcessNetwork`, `Process`, `Fifo`, `InPort`, and `Outport` are derived from class `Id`, and thus the identification is supported for all these classes. In addition the classes `ProcessNetwork` and `Process` have a member function `type` which returns the class name of the object, represented by a null-terminated character string. The type is a property of the class, and is thus identical for every instance of a class

Programs Program 2.1.5 and Program 2.1.7 show examples of the usage of the member functions `fullName` and `type`, to print information about the producer and consumer processes before they terminate. The output of this example is as follows.

```
Consumer started
Producer started
Producer pc.producer: 1000 values written.
Consumer pc.consumer: 1000 values read.
```

The identification of an object is set via the first argument of its constructor. In the implementation of the constructor, the Id is passed to the constructor of the base class. In the constructor also the identification of the member objects must be set. In case of a process these are the input and output ports. Setting the Id of the member objects is done in the same way, namely via the first argument of the constructor. To ease the creation of an Id for the members, the member function `id` is available. This function takes a name as an argument and creates an Id with this name, and the current object as a parent. This Id is then passed to the constructor of the member object.

Identification is mandatory for all ports, fifos, processes and process networks. The implementation of the `type` function is also mandatory. This member function is a pure virtual function which will generate a compile error if the implementation is omitted.

2.2.2 Communication

Scalars

Processes communicate with each other via uni-directional fifo channels. A fifo has one input end and one output end, i.e., there is exactly one process that writes to the channel and there also is exactly one process (possibly the same) that reads values from the fifo. From the process point of view we have input ports which transfer data from the fifo to the process by *reading values*, and output ports which copy data from the process to the fifo by *writing* values. The syntax for reading values is:

```
read(p, x)
```

This statement reads a value from input port `p` and stores this value in variable `x`. The syntax for writing values is:

```
write(q, y)
```

This statement writes the value of variable `y` to output port `q`.

In YAPI, channels are strictly typed, i.e., one can only communicate one type of value via a fifo. In both the read and the write statement the type of the port must be identical to the type of the fifo and to the type of the variable.

Vectors

In addition to reading and writing of scalars, one can also read or write vectors. We call these functions *vector* read and *vector* write. The vector read and write will read or write the specified number of items in one call. The normal read and write functions are also referred to as *scalar* read and write. The syntax for reading a vector of values is:

```
read(p, X, m)
```

This statement reads `m` values from input port `p` and stores these values consecutively in array `X` in array elements `X[0]` up to `X[m-1]`. The syntax for writing an array of values is:

```
write(q, Y, n)
```

This statement writes `n` consecutive values of array `Y` from array elements `Y[0]` up to `Y[n-1]` to output port `q`. In vector read and write statements the type of the port must be identical to the type of the array element. For instance, if the ports are of type integer then `X` and `Y` must be arrays of integers. Note that the number of communicated values does not have to be known at compile-time, i.e., the value of variables `m` and `n` is determined at run-time.

To illustrate the use of vector read and write operations, we modify the Producer-Consumer example. First we introduce a vector write operation in the producer. To this end, we introduce an array `Y` of size `n` to which we assign the values that are to be written with a vector write operation.

Program 2.2.3 Vector Implementation of the Producer

```
void Producer::main()
{
        const int n = 1000;

        write(out, n);

        int Y [n];

        for (int i=0; i<n; i++)
        {
                Y[i] = i;
        }

        write(out, Y, n);

        std::cout << type() << " "
                  << fullName() << ": "
                  << n << " values written"
                  << std::endl;
}
```

Similarly, we introduce a vector read operation in the consumer. To this end, we introduce an array `X` of size `n`. Note that the value of `n` is not known to the consumer at compile time. Therefore, we allocate and free memory space at run time using the `new` and `delete` operations.

Program 2.2.4 Vector Implementation of the Consumer

```
void Consumer::main()
{
        int n;

        read(in, n);

        int* X = new int [n];

        read(in, X, n);

        for (int i=0; i<n; i++)
        {
                assert(i==X[i]);
        }

        delete [] X;

        std::cout << type() << " "
                  << fullName() << ": "
                  << n << " values read"
                  << std::endl;
}
```

Note that the vector implementation of the producer (consumer) can be combined with the scalar implementation of the consumer (producer), since the interfaces of the processes have not changed. Only their internal implementations have changed. For example, the producer can be vector based having internally an array variable, while the consumer can be scalar based having internally a single variable. Hence, this example shows the purpose of the vector read and write operations, that is, to enable the decoupling of the data structures of the processes without introducing the need for additional read and/or write function calls.

Furthermore, note that it is up to the actual implementation of the vector read and write operations how the `n` consecutive values are communicated. For instance, when the room in the fifo is sufficient, the vector write may copy all values in one burst. Only if there is insufficient room in the fifo (e.g. when small fifos are used), the communication will take place at the smaller grain size, but this is hidden from the application designer.

Non-determinism

The process networks we discussed so far are deterministic, i.e., for given input they produce the same output, independent of how the execution of the processes

is scheduled. This property allows the application designer to plug a process or process network into an application and to execute it without the concern of scheduling the processes. However, there are applications, for instance interactive applications, that are not deterministic. To be able to model these applications as process networks, we introduce the synchronization operation *select*. The purpose of the select operation is to synchronize the calling process with non-deterministic neighboring processes.

A process that calls a select operation synchronizes with another process when the other process has committed to communicate the number of tokens requested by the calling process. In general, a select operation can take an arbitrary number of ports as arguments. With each port we associate a positive integer argument that indicates that the port must be willing to communicate that number of tokens in order to be a candidate for selection. This number is called the requested number of tokens. The default value for the requested number is one. Optionally, one can specify other values but in that case these values have to be specified for all ports in the select operation. We refer to a select operation without and with optional arguments as *scalar* and *vector* select operation, respectively. The syntax for a scalar select operation between two ports is

```
select(port0, port1)
```

The syntax for a vector select operation between two ports is

```
select(port0, n0, port1, n1)
```

An input port is a candidate for selection if and only if it holds that the requested number of tokens is smaller than or equal to the number of tokens that the corresponding output port has committed to produce plus the number of tokens in the fifo. An output port is a candidate for selection if and only if it holds that the requested number of tokens is smaller than or equal to the number of tokens that the corresponding input port has committed to consume minus the number of tokens in the fifo. A select operation returns an integer value that identifies the number of the selected port assuming that the ports are consecutively numbered starting from zero. So the two select operations mentioned above return 0 or 1. Note that a select operation does not terminate, i.e., its execution is suspended, until at least one of the mentioned ports is a candidate for selection.

To illustrate the use of select operations, we extend the Producer-Consumer example to a Producer-Filter-Consumer example. The function of the filter is to multiply the incoming integer values with an integer coefficient and to output the results. We

assume that the filter coefficients are produced by another process. The number of coefficients produced by this process nor the rate of production are known. Note that also the time at which they are produced is not known because each process has its own notion of time. In the filter we may read a new coefficient before receiving a new integer value. This is done only if the select operation has selected the port from which the coefficients are read, i.e., when the producer of the coefficients has committed to produce at least one other coefficient. If the select operation has selected the other port then we process the next value. The select operation blocks until it can select a port, i.e., until the other processes have committed to communicate a coefficient or a value.

Program 2.2.5 Filter Process Declaration

```
class Filter : public Process
{
public:
        Filter(const Id& n, In<int>& i0,
               In<int>* i1, Out<int>& o);
        const char* type() const;
        void main();

private:
        InPort<int>  in0; //input port for coefficients
        InPort<int>  in1; //input port for values
        OutPort<int> out; //output port for values
};
```

Program 2.2.6 Filter Process Implementation

```
void Filter::main()
{
        int value;
        int coefficient = 1;

        while (true)
        {
                if (select(in0, in1) == 0)
                {
                        read(in0, coefficient);
                }
                read(in1, value);
                write(out, coefficient*value);
        }
}
```

Note that if the application designer adds an else clause to the if statement and moves the read and write operation of the values into this clause, then the filter may 'skip' coefficients, i.e., some coefficients may never be used in a multiplication. Such a decision affects the functionality. Therefore it must be made explicit by the application designer.

Finally, note that the selection algorithm that selects a port when there is more than one candidate cannot be controlled by the application designer. The application designer must assume that the selection is a non-deterministic choice between the candidate ports. The reason for this is that in general the status of the ports, i.e., whether a port is a candidate or not, depends on the scheduling of the processes. Vice versa the scheduling of the processes depends on the selection of the ports. The scheduling algorithm of the processes is a design decision taken at a later stage by the system designer based on additional information such as computation and communication delays, resource sharing, timing constraints, etc. Similarly the selection algorithm is a design decision taken by the system designer when mapping an application onto an architecture. These design decision cannot be addressed with YAPI. The application designer has to program the application in such a way that the application is functionally correct independent of the implemented scheduling and selection algorithms.

2.2.3 Fifo Sizes

Normally, a fifo is initialized as follows:

```
fifo(id("fifo"))
```

This initializes the fifo without specifying the size. Normally, the sizes of fifos have no effect on the functionality of the application, unless they are too small which may cause deadlock. We speak of deadlock when none of the processes can make progress and none of the processes has terminated.

We distinguish between two cases of deadlock. In the first case, all processes are blocked by either a read or a select operation. In that case the fifo sizes do not affect the occurrence of deadlock. Instead you should analyze the communication behavior in your process network since the number of produced tokens is smaller than the number of tokens to be consumed. In the second case, one or more processes are blocked by a write operation. In that case our run-time environment attempts to automatically increase the size of the corresponding fifo such that the write operation can be completed and deadlock is avoided. This will fail if the maximum fifo size is reached. The run-time environment by default allocates 8 kilobytes of

memory for each fifo. Hence, the maximum size of a fifo is equal to 8192 bytes divided by the number of bytes of one token.

To support deadlock analysis, fifos can be declared with two optional arguments, namely the minimum size and the maximum size. A fifo with a minimum size of 100 tokens is initialized as follows:

```
fifo(id("fifo"), 100)
```

In this case the run-time environment may still decide to choose a larger size, but not smaller than 100. Note that the specification of a minimum size is functional because it specifies constraints to prevent deadlock. This is needed if the environment does not automatically increase fifo sizes at run-time. If the minimum size exceeds the default maximum size of the run-time environment, then the maximum size of the run-time environment is changed into the minimum size that has been specified by the application designer.

A fifo with at least 100 tokens and at most 200 tokens is initialized as follows:

```
fifo(id("fifo"), 100, 200)
```

In this case the run-time environment chooses a fifo size between 100 and 200 tokens. Note that the specification of a maximum size is not functional but that it adds implementation constraints. The application designer can use the maximum fifo size to determine the minimum fifo size at which the process network is free of deadlock. To this end, typically the minimum and maximum size are equal such that the fifo size is fixed. Once the application designer has determined the minimum size, the maximum size can be discarded to obtain maximum implementation freedom.

2.2.4 Multicast

A fifo can be connected to multiple input ports. In that case, the data that is written into the fifo will be delivered to all input ports, i.e., all input ports receive all the data that is written by the output port. Multicast is automatically implemented if a fifo is given as argument to multiple process or process network constructors. The code fragment below gives an example.

```
fifo(     id("fifo")),
producer( id("producer"),  fifo),
consumer1(id("consumer1"), fifo),
consumer2(id("consumer2"), fifo)
```

Similarly, multicast is automatically implemented if you give an input port of a process network as argument to multiple constructors of processes and process networks.

Please note that each output port can be connected to at most one other object. Hence, it is not possible to connect an output port to a fifo and to an output port of the encapsulating process network in order to use data both inside and outside of that process network. In such cases, we recommend to use a special output process that reads data from a multicast fifo and writes it to an output port of the process network. The internal processes and process networks can read from that same multicast fifo.

A select operation (Section 2.2.2) on an output port that is connected to multiple input ports can select that output port if one of the connected input ports requests sufficient tokens. So it has OR-semantics.

The full name of a multicast fifo is extended with an additional index between square brackets in the status and workload tables; see Section 2.3.1 and Section 2.4. Each output and input port pair gets a unique index. The indices are consecutively numbered starting from zero.

2.3 Run-Time Environment

A run-time environment is an environment that can execute process networks. In this section we present a run-time environment which is designed for functional simulation of process networks. The run-time environment can be activated by calling the function `start` which is declared in the include file `yapi.h`, see for example Program 2.1.1.

2.3.1 Deadlock

When none of the processes can make progress and none of the processes has terminated, we speak of deadlock. In case of deadlock the run-time environment terminates the application and displays information about the history and status of the processes and fifos in order to support the programmer with the debugging of the application.

As an example we modify the Producer-Consumer program by adding a fifo from the consumer to the producer and by letting both the producer and the consumer read from an empty fifo. To save space we leave out the declarations in the header files and only show the implementations in the source files. The process network is graphically depicted in Figure 2.3.

Figure 2.3: Modified Producer-Consumer example suffering from deadlock.

Program 2.3.1 Process Network Implementation

```
#include "pc.h"

PC::PC(const Id& n) :
        ProcessNetwork(n),
        fifo0   (id("fifo0")),
        fifo1   (id("fifo1")),
        producer(id("producer"), fifo1, fifo0),
        consumer(id("consumer"), fifo0, fifo1)
{ }
```

Program 2.3.2 Producer Process Implementation

```
#include "producer.h"
#include <iostream>

Producer::Producer(const Id& n, In<int>& i, Out<int>& o) :
        Process(n),
        in( id("in"), i),
        out( id("out"), o)
{ }

void Producer::main()
{
        const int n = 1000;

        write(out, n);

        for (int i=0; i<n; i++)
        {
                write(out, i);
        }

        int x;
        read(in, x);
        write(out, x);

        std::cout << type() << " "
                  << fullName() << ": "
                  << n << " values written"
                  << std::endl;
}
```

Program 2.3.3 Consumer Process Implementation

```
#include "consumer.h"
#include <assert.h>
#include <iostream>

Consumer::Consumer(Id n, In<int>& i, Out<int>& o) :
        Process(n),
        in( id("in"), i),
        out( id("out"), o)
{ }

void Consumer::main()
{
        int n;

        read(in, n);

        for (int i=0; i<n; i++)
        {
                int j;
                read(in, j);
                assert(i==j);
        }

        int x;
        read(in, x);
        write(out, x);

        std::cout << type() << " "
                  << fullName() << ": "
                  << n << " values read"
                  << std::endl;
}
```

The additional code in the producer and consumer causes deadlock since the consumer waits for input on fifo 0 and the producer waits for input on fifo 1. The runtime environment detects the deadlock and displays a list of the blocked processes and a fifo status table, as shown below. The list of blocked processes shows that the consumer process `pc.consumer` and the producer process `pc.producer`, are blocked by read actions on fifos `pc.fifo0` and `pc.fifo1`, respectively.

```
YAPI > Deadlock, since all processes have blocked:
Blocked processes:
---------------------------
|Process     by   on Fifo |
|pc.consumer read pc.fifo0|
|pc.producer read pc.fifo1|
---------------------------


Fifo Status:
------------------------------------------------------------------------------
|         min  max size room data Wtokens Rtokens Wpend Rpend Wneed Rneed|
|pc.fifo0   1 2048  128  128    0    1001    1001     0     1     0     0|
|pc.fifo1   1 2048  128  128    0       0       0     0     1     0     0|
------------------------------------------------------------------------------
```

The fifo status table has 12 columns, and lists the state of all fifos in the process network at the moment when the deadlock occurred. Per fifo it lists the name, the minimum and maximum size, the actual size, the free space in the fifo (room), the number of tokens in the fifo (data), the number of tokens that have been written (Wtokens), the number of tokens that have been read (Rtokens), the number of tokens in a pending write (Wpend), the number of tokens in a pending read (Rpend), the number of tokens in a pending select on an output port (Wneed), and the number of tokens in a pending select on an input port (Rneed). Especially the last four colums are important for detecting the cause of a deadlock. A non-zero number of pending tokens indicates that a read or write call has been issued, but the call has not terminated yet because not all tokens could be communicated, i.e., the process is blocked.

In this example Rpend of fifos `pc.fifo0` and `pc.fifo1` is 1, which is an indication that a process is waiting for one token to be read. The list of blocked processes can be used to find the corresponding process, which are `pc.consumer` and `pc.producer` in this example.

From the table we can furthermore derive that both fifos have an actual size of 128 values. The fifos contain no data, and thus, there is room for 128 values. 1001 values have been written and 1001 values have been read.

When deadlock has been detected and the deadlock information has been printed, the `start` function will return and the program can continue normally.

The use of a select operation can introduce deadlock. As an example we take a select operation on a single input port with a request for ten tokens. A write operation on the corresponding output port of ten or more tokens does not introduce deadlock because the number of committed tokens is larger than or equal to the number of requested tokens. A write operation on the corresponding output port of less than ten tokens introduces deadlock if and only if the number of committed

tokens plus the number of tokens in the fifo is smaller than the number of requested tokens and larger than the fifo size.

As a second example we take a select operation on a single output port with a request for ten tokens. A read operation on the corresponding input port introduces deadlock if and only if the number of committed tokens minus the number of tokens in the fifo is smaller than the number of requested tokens and larger than zero.

The above-mentioned deadlock for the select operation on the input port can be avoided either by increasing the size of the fifo or by increasing the number of committed tokens in the write operation. The above-mentioned deadlock for the select operation on the output port can be avoided only by increasing the number of committed tokens in the read operation.

2.3.2 Termination

A process network terminates when all processes terminate. In that case the `start` function will return and the program can continue normally.

When none of the processes can make progress and one or more processes have terminated, we assume that we are finished. In order to find out which processes terminated and which processes did not terminate we print the process status table and fifo status table as in the case of deadlock as explained in the previous section.

2.3.3 Stack Size and Stack Overflow

In the YAPI run-time environment a separate stack space is allocated for each process in the process network. This stack is used to store the intermediate results of a process. All function calls from the main member function will be done on the stack, which includes the storage for local (automatic) variables. Intermediate results in expressions may also be stored on the stack if insufficient registers are available. The private member variables of a process and memory that is obtained via `malloc` or `new` are not allocated on the stack.

To limit the total amount of stack space to a reasonable amount the stack space of each process is fixed to 64 kilobytes. This should suffice for the individual stack of most processes as well as for the total stack due to the total number of processes. However, for some processes this stack size might be insufficient and may result in a stack overflow. In the YAPI run-time environment a stack overflow will not result in an automatic increase of the stack space, but will result in an access violation as a result of a protected memory space that is inserted between the stacks of two

processes. The access violation causes the program to be killed and a core dump to be generated. Next, a debugger (gdb or ddd) can be used to find the location of the error. Usually this happens on a source line where a large data structure such as an array is accessed. Note that this form of stack protection is not available in the CYGWIN version of the YAPI run-time environment.

Of course it is better to avoid the above-mentioned stack overflows and associated core dumps. Since it is difficult to determine the required stack space of a process we suggest that you simply avoid that large data stuctures are allocated on the stack. For video applications this typically means that you should not declare complete video frames, and at most a few video lines as local (automatic) variables in your functions. If you need large data structures in your application it is best to allocate these dynamically on the heap, using *malloc* or *new*. Another option is to declare these variables as private members in your process.

2.4 Workload Analysis

Two important properties of an application are its *communication workload* and its *computation workload* [20]. The communication workload is a measure for the amount of communication in an application, given the input data. The computation workload is a measure for the amount of computation in the application, given the input data. In the YAPI run-time environment the communication workload is measured by counting the number of tokens that are traveling through the fifos. The communication workload can be printed by calling the following function after the application has ended:

```
void printCommunicationWorkload(
                const ProcessNetwork& n,
                std::ostream& o = std::cout);
```

The output for an example application with five fifos is shown below. The first column lists the full names of the fifos. The column `Wtokens` lists the number of tokens that have been written to the fifo. The column `Wcall` lists the number of write calls, and the column `T/W` lists the average number of tokens per write call. Similarly the columns `Rtokens`, `Rcalls`, and `T/R` give the values for the read calls.

```
Communication Workload:
-------------------------------------------------
|          Wtokens Wcalls T/W Rtokens Rcalls T/R|
|pfc.fifo1       1      1   1       1      1   1|
|pfc.fifo2      10     10   1       5      5   1|
|pfc.fifo3      10     10   1      10     10   1|
|pfc.fifo4    1003   1003   1    1003    103   9|
|pfc.fifo5    1003   1003   1    1003   1003   1|
-------------------------------------------------
```

YAPI also supports the measuring and printing of computation workload. To measure the computation workload, the C/C++ code of the processes must be annotated with *execute* statements. An execute statement is a call of the protected member function `execute` of class `Process`. The argument of the execute statement is a symbolic instruction that represents the workload that is associated with a certain code fragment.

```
void Process::execute(const char* instr);
```

In the YAPI run-time environment the number of execute calls are counted. These numbers can be printed with the function `printComputationWorkload`.

```
void printComputationWorkload(
                  const ProcessNetwork& n,
                  std::ostream& o = std::cout);
```

The table below shows how the output may look like. The first column lists the full names of the processes, the second column lists the symbolic instruction names, and the third column lists the number of calls that occurred during execution. The processes `ppc.prd1` and `ppc.prd2` have executed the instruction `produce` 100000 times, `ppc.cons` has executed instruction `start` and `stop` once, and has executed instruction `consume` 200000 times.

```
Computation Workload:
-----------------------------
|Process  Instruction  Count|
|ppc.prd1                   |
|         produce     100000|
|ppc.prd2                   |
|         produce     100000|
|ppc.cons                   |
|         start       1     |
|         consume     200000|
|         stop        1     |
-----------------------------
```

By associating execution times with the instructions, and by multiplying these execution times with the instruction counts, one can get a rough estimate of the total execution time of the application on a certain platform.

2.5 Visualization of Process Networks

When large process networks are created is is difficult to obtain a good overview of the network. In such a case a graphical output of the network, made via the *dotty* tools can be useful.

2.5.1 What is Dotty?

The dotty format is a textual format that can be used to represent graphs. The dotty tools can then be used to create a layout of the graph and output it as a postscript file. The postscript file can then be used for documentation purposes. An example process network that is visualized via dotty is shown in Figure 2.4. Dotty is a part of the *Graphviz* tools, and can be obtained from http://www.research.att.com/sw/tools/graphviz . Note that dotty is available for users at the Nat.Lab. through the cadappl tree (`cadenv graphviz`).

2.5.2 How to Use Dotty

The function `printDotty` is used to print a process network in dotty format.

```
void printDotty(const ProcessNetwork& p,
                std::ostream& o = std::cout);
```

To generate a dotty file one can add a `printDotty` call to the *main* program, as shown in the example below. In this example the process network `nw` of type `MyNetwork` is created, which is then printed in dotty format in the file `nw.dot`.

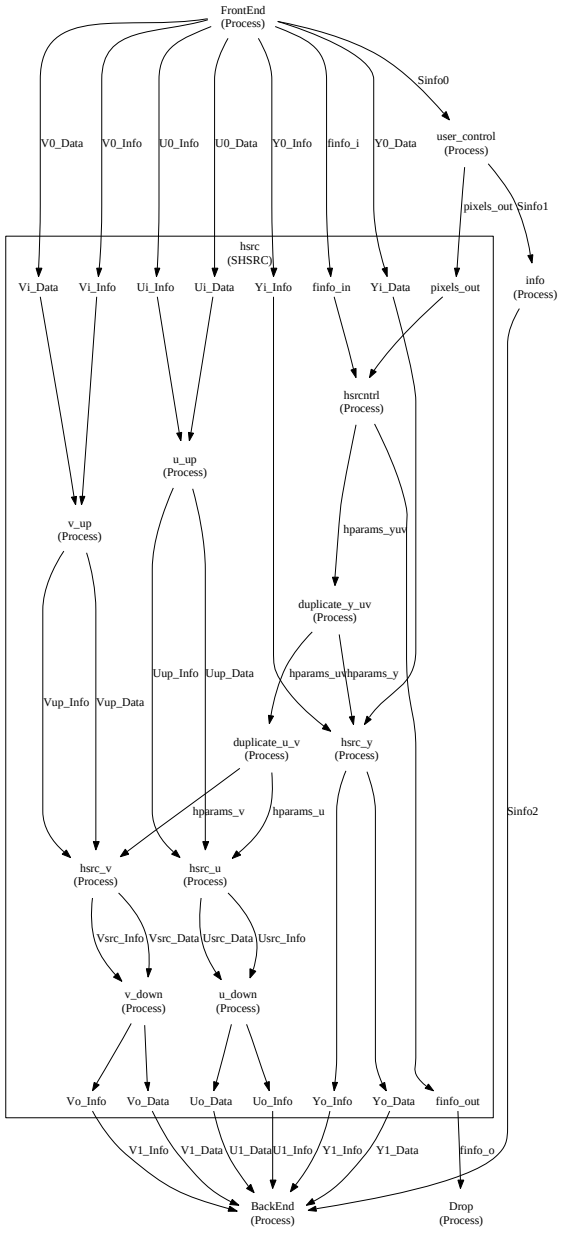

Figure 2.4: Dotty Output

Program 2.5.1 Dotty Output

```
#include "mynetwork.h"
#include "dotty.h"
#include <fstream>

main()
{
        // create the network
        MyNetwork nw(id("nw"));

        // create output file nw.dot
        std::ofstream f("nw.dot");

        // print network in dotty format
        printDotty(nw, f);
}
```

After creating the .dot file by running the program, the graphviz tools can be used to visualize the .dot file. The program `dotty` can be used to view the graph via the X environment, or the program `dot` can be used to convert the graph into postscript format, with the following command:

```
dot -Tps nw.dot -o nw.ps
```

For output in other formats we refer to the graphviz manual pages.

2.5.3 Options

The YAPI to dotty conversion has various options to control the formatting of the graph. These options are specified via a configuration file named `dotty.cfg` that must be present in the current directory or in your home directory. The configuration file is a text file in which the options are specified as `name = value` pairs. Below, the default option settings are listed.

Program 2.5.2 Default Option Settings

```
InitLevel                       = 10
PageSize                        = A4p
Ratio                           = fill
Fifo                            = false
Decorate                        = false

ProcessNetworkColor             = white
ProcessColor                    = white
PortColor                       = white
FifoColor                       = white

ProcessNetworkClusterFontColor = black
ProcessNetworkFontColor         = black
ProcessFontColor                = black
PortFontColor                   = black
FifoFontColor                   = black

ProcessNetworkClusterFontName  = Helvetica
ProcessNetworkFontName          = Helvetica
ProcessFontName                 = Helvetica
PortFontName                    = Helvetica
FifoFontName                    = Helvetica

ProcessNetworkClusterFontSize  = 20
ProcessNetworkFontSize          = 16
ProcessFontSize                 = 16
PortFontSize                    = 16
FifoFontSize                    = 16
```

The options can be chosen from the following values:

- **InitLevel**: integer.
  The number of hierarchical levels that is shown. When set to 1, only the top level prosess network is shown, and all sub-networks are shown as simple nodes. When set to 2, the contents of the top level network and its sub-networks are shown.
- **PageSize**: A4p, A4l, A3p, A3l, A2p, A2l, A1p, A1l, A0p or A0l.
  Specifies the page size and orientation. For example, A4p represents A4 portrait, A2l represents A2 landscape, etc.
- **Ratio**: fill, compress, or auto.
  Fill stretches the drawing, whereas *compress* compresses the drawing to fit it in the given pagesize.
- **Fifo**: true, or false.

If set to true, fifos are shown as diamond-shaped nodes, otherwise as normal edges.

- **Decorate**: true, or false.
  If set to true, a line is drawn that connects the fifo label with the edges. These lines are only visible in the postscript output, not in X (dotty).
- **Color**: white, lavender, gray, black, red, pink, brown, beige, orange, yellow, gold, green, cyan, blue, navy, magenta, purple, or violet.
- **Font**: Times-Roman, Helvetica, or Courier.

2.6 Programming Guidelines

2.6.1 Coding Style

Processes

The preferred coding style for the definition of process types is indicated in the template listed below. This template is also part of the release in the subdirectory containing the examples. The template for user-defined processes is listed in the files `myprocess.h` and `myprocess.cc`. A user-defined process consists of a public part, that is accessible to the environment of a process instance, and of a private part, that is only accessible to the process instances themselves. The public part consists of four consecutive sections, namely *constructors*, *destructor*, the *type member function*, and the *main member function*. The private part consists of four consecutive sections, namely *input ports*, *output ports*, *member functions*, and *member variables*.

Program 2.6.1 Process Template

```
class MyProcess : public Process
{
public:
        // constructors
        MyProcess(const Id& n, In<int>& i,
                  Out<int>& o, int x);

        // destructor
        ~MyProcess();

        // type member function
        const char* type() const;

        // main member function
        void main();
private:
        // input ports
        InPort<int>     in;

        // output ports
        OutPort<int>    out;

        // member functions
        int             f(int);

        // member variables
        int             a;
};
```

The constructor section contains one or more constructors. Each constructor has one or more arguments. The first argument is an instance name that is given when the programmer instantiates a process. The next arguments are optional; they are inputs, outputs, and parameters, respectively. More specically, the subsequent zero or more arguments are inputs that are bound to the input ports. The number, the type, and the order of inputs must correspond to the number, the type, and the order of input ports in the input port section. The next zero or more arguments are outputs that are bound to the output ports. The number, the type, and the order of outputs must correspond to the number, the type, and the order of output ports in the output port section. The final zero or more arguments are parameters that are used to initialize the member variables of the member variable section.

The destructor section contains an optional destructor. The purpose of the destructor is to deinitialize the member variables and to deallocate memory that has been allocated in the constructor. The type member function section contains one function that must return the type of the process, i.e., the class name which is in this

case “MyProcess”. The input port section contains zero or more input ports. The number, the type, and the order must correspond to those of the inputs in the constructors. The output port section contains zero or more output ports. The number, the type, and the order must correspond to those of the outputs in the constructors. The member function section contains zero or more functions which can be called from the main function. The member variable section contains zero or more member variables that can be accessed by the member functions including the main function.

The initialization of process instances is done in a constructor of the corresponding process type. The constructor of a process type contains a member initialization section and a member assignment section. The member assignment section may be used to assign a value to the member variables that are not initialized via the initializer list. The constructor cannot contain calls to member functions.

Program 2.6.2 Process Constructor Template

```
MyProcess::MyProcess(const Id& n, In<int>& i,
                     Out<int>& o, int x) :
        // member initialization
        Process(n),
        in(id("in"), i),
        out(id("out"), o)
{
        // member assignment
        a = x;
}
```

The initialization list passes the instance name of the user-defined process to the predefined base class `Process`. Furthermore the list initializes the input ports and output ports. To this end one must supply instance names as well as the binding of the ports to the inputs and outputs.

The functionality of a process type is defined in the main member function. Each process instance executes the main member function using its own state space, i.e., each process instance has its own program counter, its own memory space containing the variables as defined in the process type, its own ports, etcetera. The functionality in the main member function as well as the other member functions is described in terms of regular C or C++ constructs and the predefined YAPI functions read, write and select. Variables that are used in only one member function of a process type have to be declared locally in the scope of the specific member function. Variables that are used in more than one member function and that are not passed as arguments of the member functions have to be declared as member variables. It is important to note that one must not declare *global* or *static* variables

other than constants because these variables are shared by all process instances of a process type. Consequently, all process instances operate in parallel on the same global or static variable which will result in unpredictable behavior because the accesses on this variable are not synchronized.

Process Networks

The preferred coding style for the definition of process network types is indicated in the template listed below. This template is also part of the release in the subdirectory containing the examples. The template for user-defined process networks is listed in the files `mynetwork.h` and `mynetwork.cc`. A user-defined process network consists of a public part, that is accessible to the environment of a process network instance, and of a private part, that is only accessible to the process network instances themselves. The public part consists of two consecutive sections, namely *constructors* and the *type member function*. The private part consists of five consecutive sections, namely *input ports*, *output ports*, *fifos*, *processes*, and *process networks*.

Program 2.6.3 Process Network Template

```
class MyNetwork : public ProcessNetwork
{
public:
        // constructors
        MyNetwork(const Id& n, In<int>& i,
                  Out<int>& o, int x);

        // type member function
        const char* type() const;
private:
        // input ports
        InPort<int>     in;

        // output ports
        OutPort<int>    out;

        // fifos
        Fifo<int>       fifo;

        // processes
        MyProcess       myprocess;

        // process networks
        MyOtherNetwork  mynetwork;
};
```

The constructor section is the same as the constructor section of processes. It contains one or more constructors. Each constructor has one or more arguments. The first argument is an instance name that is given when the programmer instantiates a process network. The next arguments are inputs, outputs, and parameters, respectively. More specifically, the subsequent zero or more arguments are inputs that are bound to the input ports. The number, the type, and the order of inputs must correspond to the number, the type, and the order of input ports in the input port section. The next zero or more arguments are outputs that are bound to the output ports. The number, the type, and the order of outputs must correspond to the number, the type, and the order of output ports in the output port section. The final zero or more arguments are parameters that are passed to processes and process networks in the process and process network sections, respectively.

The type member function section contains one function that must return the type of the process network, i.e., the class name which is in this case "MyNetwork". The input port section contains one or more input ports. The number, the type, and the order must correspond to those of the inputs in the constructors. The output port section contains one or more output ports. The number, the type, and the order must correspond to those of the outputs in the constructors. The fifo section contains zero or more fifos. The process section contains zero or more processes. The process network section contains zero or more process networks.

The initialization of process network instances is done in a constructor of the corresponding process network type. The constructor of a process network type only contains a member initialization section. The member assignment section is obsolete because process network types only contain member variables that have a constructor.

Program 2.6.4 Process Network Constructor Template

```
MyNetwork::MyNetwork(const Id& n, In<int>& i,
                     Out<int>& o, int x) :
        // member initialization
        ProcessNetwork(n),
        in(id("in"), i),
        out(id("out"), o),
        fifo(id("fifo")),
        myprocess(id("myprocess"), in, fifo, x),
        mynetwork(id("mynetwork"), fifo, out)
{ }
```

The initialization list passes the instance name of the user-defined process network to the predefined base class `ProcessNetwork`. Furthermore the list initializes the input ports, output ports, fifos, processes, and process networks. To this end

one must supply instance names, the binding of the ports to the inputs and outputs, and the binding of processes and process networks to the ports and fifos. Note that it is not allowed to pass the inputs and outputs directly to the processes and process networks.

2.6.2 Design Rules

In this section we present design rules to develop reusable YAPI applications. By reuse we mean two things. Firstly, to reuse applications in the design of alternative applications. Secondly, to reuse applications in the design of alternative architectures. To illustrate these design rules we use examples from VYA (Video YAPI) [23], which is a set of design rules for applying YAPI to model video applications.

Rule 1. Introduce functional data types

The introduction of functional data types improves reusability because it makes data types explicit. As an example we mention the data type `VYApixel` that is used to represent R, G, and B as well as Y, U, and V values. One advantage is that this data type abstracts from the representation of these values in an implementation, which can be for instance `int` or `short`. Another advantage is that it is clear what data type is used such that type checking at compile-time is feasible. For example, the data types `VYAlineLength` and `VYAimageWidth` are both defined as `unsigned int`, but the former is associated with lines while the latter is associated with images. Hence, functional data types are important, especially for the definition of the interfaces of processes to increase their reusability in alternative applications.

Rule 2. Minimize latency

The latency of a process is the difference between the number of values that it has read and the number of values that it has written. In general each output value functionally depends on one or more input values, such that these input values need to be read before the output value can be computed and written. By reading the input values as late as possible, i.e., only when you need them, and by writing the output values as soon as possible, i.e., immediately after you compute them, the number of living values in the process is minimal. This minimizes the memory requirements of the process and of its environment (i.e., the fifos of the process network) if its output needs to be synchronized with other streams. Here synchronization means to match the latency of the process and the latency of the other streams by storing these streams in fifos. By minimizing the latency of the processes also the fifo sizes that are minimally needed to avoid deadlock are reduced. This increases the reusability of the process in alternative applications.

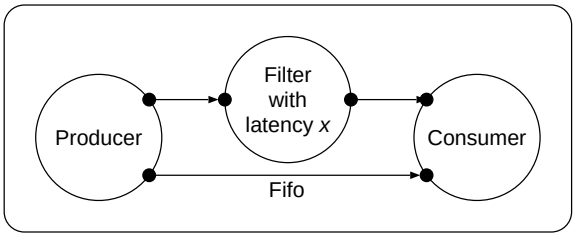

Figure 2.5: The relation between process latencies and fifo sizes.

To illustrate the relation between latency and fifo sizes we take the example of Figure 2.5. We assume that the producer alternatingly writes one value to its two ports. Similarly the consumer alternatingly reads one value from its two ports. We assume that they both start with the upper port. Due to latency x of the filter, the first read action of the consumer cannot be completed until the producer has written x values to its upper port. Hence, the fifo between the producer and the consumer must be able to store at least $x - 1$ values to prevent deadlock.

Rule 3. Use vector operations

Vector operations have been introduced to efficiently support the reading and writing of composite data types. Suppose for example that we have a process that internally has a data type for video lines which is composed out of video pixels. In VYA this would typically be written as follows

```
typedef Line VYApixel[n];
```

where variable `n` is of type `VYAlineLength`, which is defined as `unsigned int`. The value of a variable `line` of type `Line` can be read using vector operation `read(in, line, n)` or written using vector operation `write(out, line, n)` in a single operation. Hence, the vector operations abstract from how the video line is communicated in the implementation. The implementation can communicate the line pixel by pixel, in groups of pixels, or in its entirety. So the abstraction provides more implementation freedom which increases the reusabilty in alternative architectures.

Rule 4. Avoid "resource" sharing

Ports, fifos, processes, and process networks are functional entities, they are not resources. Therefore, they must not be shared for multiple purposes. As an example we mention the interface of VYA processes and process networks which always have separate ports for Y, U, and V values of type `VYApixel`. In 4:2:2 format these streams are often multiplexed in one YUYV stream, but this is a de-

sign choice which must be made explicit in a mapping step if one maps the three YAPI ports for Y, U, and V, onto a single port in the architecture.

Another example of “resource” sharing is RGB to YUV conversion and vice versa. Although the functions for both cases might be quite similar apart from the coefficients, the interfaces for both cases are different. For RGB to YUV conversion we have R, G, and B inputs and Y, U, and V outputs. For YUV to RGB we have Y, U, and V inputs and R, G, and B outputs. Do not try to model these two functions in a single process because the resulting process is not reusable for others that need only one of the two functions. Moreover, the resulting process lacks the concurrency that two separate processes would have had. If one uses both processes in the same process network, then it is possible in the mapping step to map both processes onto a single processor mutually exclusive in time. One should not attempt to model this design decision in YAPI.

2.7 Reference

2.7.1 Function `read`

Program 2.7.1 Read

```
template<class T>
void read(InPort<T>& s, T& t)

template<class T>
void read(InPort<T>& s, T* p, unsigned int n)
```

The function `read` (in file `port.h`) is an overloaded template function that consumes data from a given input port and stores this data in a given variable. The type of the input port must match with the type of the variable. This means that either both types are identical, or the variable is an array of the type of the input port. In the latter case one must provide a third argument to the read function which indicates the number of elements to be read.

2.7.2 Function `write`

Program 2.7.2 Write

```
template<class T>
void write(OutPort<T>& s, const T& t)

template<class T>
void write(OutPort<T>& s, const T* p, unsigned int n)
```

The function `write` (in file `port.h`) is an overloaded template function that copies[1] data contained in a given variable to a given output port. The type of the output port must match with the type of the variable. This means that either both types are identical, or the variable is an array of the type of the output port. In the latter case one must provide a third argument to the write function which indicates the number of elements to be written.

[1] A shallow copy is made, i.e. if the type to be copied is a struct containing pointers, only the struct is copied, not data that is pointed to.

2.7.3 Function `select`

Program 2.7.3 Select

```
inline unsigned int select(SelectList& l);

template<class T0,class T1>
unsigned int select(InPort<T0>&  p0, InPort<T1>&  p1)

template<class T0,class T1>
unsigned int select(InPort<T0>&  p0, OutPort<T1>& p1)

template<class T0,class T1>
unsigned int select(OutPort<T0>& p0, OutPort<T1>& p1)

template<class T0,class T1>
unsigned int select(InPort<T0>&  p0, unsigned int n0,
                    InPort<T1>&  p1, unsigned int n1)

template<class T0,class T1>
unsigned int select(InPort<T0>&  p0, unsigned int n0,
                    OutPort<T1>& p1, unsigned int n1)

template<class T0,class T1>
unsigned int select(OutPort<T0>& p0, unsigned int n0,
                    OutPort<T1>& p1, unsigned int n1)
```

The function `select` (in file `select.h`) is an overloaded template function that selects one of the given ports that is able to communicate the given requested number of tokens. The default value for the number of tokens is one. An input port can be selected if and only if it holds that the requested number of tokens is smaller than or equal to the number of tokens that the corresponding output port has committed to produce plus the number of tokens in the fifo. An output port can be selected if and only if it holds that the requested number of tokens is smaller than or equal to the number of tokens that the corresponding input port has committed to consume minus the number of tokens in the fifo.

To save space we have restricted the listed select operations to those that have two ports as arguments. YAPI also supports three and four ports as arguments. From YAPI Release 1.1 a select can be performed on a SelectList class that can hold an arbitrary number of ports. This is useful when a select on more than four ports is needed. For more details on SelectList see Section 2.7.13.

2.7.4 Class `Id`

Program 2.7.4 Id (id.h)

```
class IdBase
{
public:
    IdBase(const IdBase& i);
    IdBase(const char* n, IdBase* p);
    virtual ~IdBase();

    IdBase& operator+(char c);
    IdBase& operator+(const char* str);
    IdBase& operator=(const IdBase& i);

    IdBase*     parent() const;
    const char* name() const;
    const char* fullName(char* buf=0) const;

    operator const char*() const;

    IdBase id(const char* n);

private:
    char*   nm;  // name
    IdBase* pa;  // parent
};

class Id: public IdBase
{
public:
    Id(const char*, Id*);
    Id(const IdBase&);
    ~Id();
};

Id id(const char*);
```

The class `Id` (in file `Id.h`) provides a public interface to the class `IdBase`. The class `IdBase` is a base class of the classes `ProcessNetwork`, `Process`, `Fifo` `InPort`, and `OutPort`, but due to private inheritance it is not accessible.

The class `Id` has two constructors. The copy constructor of class `Id` is used to set the Id of member objects. In this case the member function `id` of class `IdBase` is called to create the Id for member objects. The `id` member function returns an IdBase with the given instance name and the object itself (this) as a parent. The

non-member function `id` is used to create an Id for the root process network. This function returns an Id with the parent set to the null pointer. The constructor of class `Id` with two arguments is used to set the Id of the root process network.

The member function `parent` returns the parent object, represented by a pointer to an IdBase. The member function `name` returns the instance name of the object, represented as a null-terminated character string. The member function `fullName` returns the path name of the object within the network hierarchy. The full name is a null-terminated character string consisting of a concatenation of the names of all objects in the path, starting from the root object. The names in the path are separated by dots. If the function `fullName` is provided with the null pointer (default argument), the result will be placed in an internal static area, which will be preserved until the next call of `fullName`. Otherwise the result will be stored in the given array of characters. Finally, operator functions + and = are available.

2.7.5 Class `In<T>`

Program 2.7.5 In (io.h)

```
template<class T>
class In
{
public:
    virtual void read(T& value) = 0;
    virtual void read(T* p, unsigned int n) = 0;

    virtual Connector* connector() = 0;
};
```

The abstract template class `In<T>` (in file `io.h`) is a base class for the classes `InPort<T>` and `Fifo<T>` that provides a function `read`. The read function implements the YAPI function read.

2.7.6 Class `Out<T>`

Program 2.7.6 Out (io.h)

```
template<class T>
class Out
{
public:
    virtual void write(const T& value) = 0;
    virtual void write(const T* p, unsigned int n) = 0;

    virtual Connector* connector() = 0;
};
```

The abstract template class `Out<T>` (in file `io.h`) is a base class for the classes `OutPort<T>` and `Fifo<T>` that provides a function `write`. The write function implements the YAPI function write.

2.7.7 Class `Fifo<T>`

Program 2.7.7 Fifo (fifo.h)

```
template<class T>
class Fifo : private FifoImplT<T>,
             public In<T>,
             public Out<T>
{
public:
    Fifo(const Id& n);
    Fifo(const Id& n, unsigned int lo);
    Fifo(const Id& n, unsigned int lo,
         unsigned int hi);

    void read(T& t);
    void read(T* p, unsigned int n);

    void write(const T& t);
    void write(const T* p, unsigned int n);

    Connector* connector();
};
```

The template class `Fifo<T>` (in file `fifo.h`) inherits from the classes `In<T>`,

and `Out<t>`. One can instantiate fifos of arbitrary types. At the time of instantiation one must provide an instance name, and optionally a minimum size, or a minimum size and a maximum size.

The fifo size has to be specified in terms of the number of elements of the fifo type. The fifo size times the number of bytes that is required to represent the fifo type gives the memory requirements in bytes. The run-time environment instantiates a fifo which has a size between the minimum and maximum size. If the maximum size is not given then it is assumed that the fifo size may be arbitrarily large. If, in addition, the minimum size is not given then it is assumed that the fifo size may be an arbitrarily small non-negative integer.

The minimum size is required to prevent deadlock in the application. The maximum size is required to cause deadlock in the application in order to analyze the occurrence of deadlock. The programmer can set the exact fifo size by setting both the minimum and maximum size to the desired size. Once the programmer has analyzed the deadlock behavior, it is a good programming practice to specify the minimum size to prevent deadlock and omit the maximum size to allow maximum implementation freedom.

The member function `connector` returns a `this` pointer, since `Connector` is a base class of class `Fifo<T>`.

2.7.8 Class `InPort<T>`

Program 2.7.8 InPort (port.h)

```
template<class T>
class InPort : public InPortImplT<T>, public In<T>
{
public:
    InPort(const Id& n, In<T>& i);

    void read(T& value);
    void read(T* p, unsigned int n);
};
```

The template class `InPort<T>` (in the file `port.h`) inherits from the class `IdBase` (via the subclass `InPortImpT<T>`), and `In<T>`. It inherits the function `read` which is called by the YAPI function read. One can instantiate input ports of arbitrary types in order to define the interface of a process. At the time of instantiation one must provide either a fifo or an input port of the encapsulating process network.

The functions `name` and `fullname` are also available. The function `name` returns the instance name of the InPort and the function `fullname` returns the path name of the InPort within the network hierarchy..

2.7.9 Class `OutPort<T>`

Program 2.7.9 OutPort (port.h)

```
template<class T>
class OutPort : public OutPortImplT<T>, public Out<T>
{
public:
    OutPort(const Id& n, Out<T>& o);

    void write(const T& value);
    void write(const T* p, unsigned int n);
};
```

The template class `OutPort<T>` (in file `port.h`) inherits from the class `IdBase` (via the subclass `OutPortImpT<T>`) and `Out<T>`. It inherits the function `write` which is called by the YAPI function write. One can instantiate output ports of arbitrary types in order to define the interface of a process. At the time of instantiation one must provide either a fifo or an output port of the encapsulating process network.

The functions `name` and `fullname` are also available. The function `name` returns the instance name of the OutPort and the function `fullname` returns the path name of the OutPort within the network hierarchy..

2.7.10 Class `Process`

Program 2.7.10 Process (process.h)

```
class Process : private ProcessImpl
{
public:
    Process(const Id& n);

protected:
    Id id(const char*);
    const char* name() const;
    const char* fullName() const;
    virtual const char* type() const = 0;

    virtual void main() = 0;
    void execute(const char* i);
};
```

The class `Process` (in file `process.h`) is a base class for user-defined process types. The class inherits from the class `IdBase` via the subclass `ProcessImpl`. The member functions `id`, `name`, and `fullName` make the corresponding member functions of the class `IdBase` accessible to the class `Process` and its subclasses. At the time of instantiation one must provide an instance name.

Furthermore, subclasses of `Process` must define a function `type` that must return the name of the subclass. In addition, they must define a function `main` that contains the functionality of the process type. A user-defined process type has input and output ports through which it communicates. The main function can read from the input ports via the function read and it can write to the output ports via the function write. To measure the computation workload, the main function of processes can be annotated with *execute* statements. An execute statement is a call of the protected member function `execute`. The argument of the execute statement is a symbolic instruction that represents the workload that is associated with a certain code fragment.

2.7.11 Class `ProcessNetwork`

Program 2.7.11 ProcessNetwork (network.h)

```
class ProcessNetwork : private NetworkImpl
{
public:
    ProcessNetwork(const Id& n);

protected:
    Id id(const char*);
    virtual const char* type() const = 0;
};
```

The Class `ProcessNetwork` (in file `network.h`) is a base class for user-defined process network types. The class inherits from the class `IdBase` via the subclass `NetworkImpl`. The member function `id` makes the member function `id` of the class `IdBase` accessible to the class `ProcessNetwork` and its subclasses. At the time of instantiation one must provide an instance name. Subclasses of `ProcessNetwork` must define a function `type` that must return the name of the subclass.

A user-defined process network has input and output ports through which it communicates. Furthermore, it has a number of internal fifos, processes, and process networks. The fifos are bound to the input and output ports of the internal processes and the internal process networks, and to the input and output ports of the process network itself.

2.7.12 Class RTE

Program 2.7.12 RTE (apiRte.h)

```
class RTE
{
public:
    RTE ()  { r = ::newRte();};
    ~RTE () { delete r;};

    void start (ProcessNetwork& n);

    void setOutStream(std::ostream& o);
    void setErrorStream(std::ostream& e);
    std::ostream& getOutStream();
    std::ostream& getErrorStream();

private:
    Rte* r;
};
```

The class RTE is derived from a specific run time environment that controls the execution of the yapi network. The execution is started by calling the start function of this class. Furthermore the user can change the direction of the info and error messages by setting the out and error stream. By default these streams are directed to cout and cerr.

2.7.13 Class `SelectList`

Program 2.7.13 SelectList (select.h)

```
class SelectList : private SelectImpl
{
public:
    SelectList (const Id& n);

    template<class T>
    void pushBack(const InPort<T>& p,
                  unsigned int n=1);
    inline void pushBack(const InPortImpl& p,
                         unsigned int n=1);

    template<class T>
    void pushBack(const OutPort<T>& p,
                  unsigned int n=1);
    inline void pushBack(const OutPortImpl& p,
                         unsigned int n=1);

    inline unsigned int select ();
};
```

The class `SelectList` is a container class for ports that can be used as argument for the select function, as described in the `function select` section. The return value of the select member function is an index that indicates which port is selected. The ports in the SelectList container are ordened. The first port that is pushed on the list with the pushBack member function referenced with index 0. The next port that is pushed on the list with the pushBack is referenced with index one, etc. At least one port must be pushed on the list before the select member is called, otherwise the behaviour of the select call is undefined.

2.7.14 Function `start`

Program 2.7.14 Start (baseRte.h)

```
void start(const ProcessNetwork& n);
```

Note, this function has become obsolete. The member function `start` of class RTE (in file `apiRte.h`) is preferred.

The function `start` (in file `baseRte.h`) is used to start the execution of the

instantiated process network. The function `start` starts all processes of the specified process network, and then waits until they have terminated or blocked. A process terminates when it reaches the end of main. One cannot explicitly start, stop, suspend, or resume individual processes. Process control can only be done via reads and writes on input and output channels.

2.7.15 Function `printWorkload`

Program 2.7.15 Workload (workload.h)

```
void printWorkload(const ProcessNetwork& n,
                   std::ostream& o = std::cout);
void printComputationWorkload(
                   const ProcessNetwork& n,
                   std::ostream& o = std::cout);
void printCommunicationWorkload(
                   const ProcessNetwork& n,
                   std::ostream& o = std::cout);
```

After the execution of the process network the computation workload and the communication workload can be printed with the functions `printWorkload`, `printComputationWorkload` and `printCommunicationWorkload` (in file `workload.h`), respectively.

2.7.16 Function `printDotty`

Program 2.7.16 Dotty (dotty.h)

```
void printDotty(const ProcessNetwork& n,
                std::ostream& o = std::cout,
                std::ostream& e = std::cerr);
```

After the declaration of a process network its structure can be visualized with the function `printDotty`.

2.8 How to Compile Your Application

To compile your application program, you can make use of the template makefile shown below. This is the makefile for the producer-consumer example. Change the variable YAPIROOT to the directory where you installed YAPI, and change the variable OBJS according to your sources.

Program 2.8.1 Makefile

```
YAPI          = $(YAPIROOT)

INCLUDES      = -I$(YAPI)/include
LIBS          = $(YAPIROOT)/lib/libyapi.a

CXX           = g++
CXXFLAGS      = -g -O $(INCLUDES)

OBJS          = consumer.o \
                producer.o \
                pc.o \
                main.o

EXE           = ./run.exe

all:    $(EXE)

test:   $(EXE)
        @$(EXE)

$(EXE): $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) $(LIBS) -o $(EXE)

clean:; rm -rf $(OBJS) $(EXE)
```

Bibliography

[1] F. Balarin et al., *Hw.-sw. co-design of embedded systems: the Polis approach*, Kluwer, 1997.

[2] J.T. Buck, *Static scheduling of code generation from dynamic dataflow graphs with integer valued control signals*, Asilomar Conf. on Signals, Systems and Computers, 508-513, 1994.

[3] W. Chang, S. Ha and E.A. Lee, *Heterogeneous Simulation - Mixing Discrete-Event Models with Dataflow*, Journal of VLSI Processing **15**, 127-144, 1997.

[4] T. Grötker, R. Schoenen and H. Meyr, *PCC: A Modeling Technique for Mixed Control/Data Flow Systems*, European Design & Test Conf., 482-486, 1997.

[5] Mentor Graphics, *DSP Station User's Manual*, San Jose, CA 95131, USA.

[6] F. Boussinot and R. de Simone, *The Esterel Language*, Proc. IEEE **79**, 1293-1304, 1991.

[7] E.A. Lee and D.G. Messerschmidt, *Synchronous data flow*, Proc. IEEE **75**, 1235-1245, 1987.

[8] A.J. Martin, *An axiomatic definition of synchronization primitives*, Acta Informatica **16**, 219-235, 1981.

[9] A.J. Martin, *The Probe: An Addition to Communication Primitives*, Information Processing Letters **20**, 125-130, 1985.

[10] J.T. Buck et al., *A platform for heterogeneous simulation and prototyping*, Proc. 1991 European Simulation Conf., 1991.

[11] Cadence Design Systems, *SPW User's Manual*, Foster City, CA 94404, USA.

[12] M. von der Beeck, *A comparison of statecharts variants*, Proc. Formal Techniques in Real Time and Fault Tolerant Systems, 128-148, 1994.

[13] A.J. Martin, *Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits*, Developments in Concurrency and Communication, C.A.R. Hoare (ed.), Addison-Wesley, 1990.

[14] G. Bilsen et al., *Static scheduling of multi-rate cyclo-static DSP applications*, Proc. Workshop on VLSI Signal Processing, 137-146, 1994.

[15] E.A. Lee and T.M. Parks, *Dataflow process networks*, Proc. IEEE **83**, 773-801, 1995

[16] Gilles Kahn, *The Semantics of a Simple Language for Parallel Programming*, Proc. of the IFIP Congress 74, North-Holland Publishing Co, 1974.

[17] C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM 21, 666-677, 1978.

[18] Bart Kienhuis, Ed Deprettere, Kees Vissers, Pieter van der Wolf, *An Approach for Quantitative Analysis Of Application-Specific Dataflow Architectures*, ASAP'97, July 14-16, 1997.

[19] Stanley B. Lippman, *C++ Primer,* 2^{nd} *Edition*, Addison Wesley, 1991.

[20] Pieter van der Wolf, Paul Lieverse, Mudit Goel, David La Hei, Kees Vissers, *An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology*, in Proc. CODES'99.

[21] Pieter Struik, *SLAM User Manual, version 2.0*, Software TV Doc. No. 098, Oct 1, 1996.

[22] Jean-Yves Brunel, Erwin de Kock, Wido Kruijtzer, Arjan Kenter, Wim Smits, *Communication Refinement in Video Systems on Chip*, in Proc. CODES'99.

[23] Kees van Zon, *VYA: Applying YAPI to Video*, Philips Research USA - TN-2000-025, 2000.

Index

Index