OFFIS
INSTITUTE FOR
INFORMATION TECHNOLOGY

R&D Division Transportation
Hardware/Software Design Methodology Group

OSSS- A Library for Synthesisable System Level Models in SystemC^{TM}

The OSSS 2.2.0 Tutorial

Oldenburg 2009

The following people have contributed to OSSS:

Claus Brunzema
Ralph Görgen
Cornelia Grabbe
Uwe Grahl
Eike Grimpe
Kim Grüttner
Philipp A. Hartmann
Andreas Herrholz
Henning Kleen
Dr. Frank Oppenheimer
Philipp Reinkemeier
Andreas Schallenberg
Thorsten Schubert
Christian Stehno
Janko Timmermann

Last compiled: January 20, 2009

Contents

1 Introduction

The following document serves as a quick introduction to OSSS (**O**ldenburg **S**ystem **S**ynthesis **S**ubset) which is an extension library to SystemC^{TM}. It presents the concept and some of the basic modelling elements. The selected OSSS concepts are explained using a simple packet processor (called producer/consumer) example. However, it should give a first impression of OSSS and how it is used to model embedded hardware/software systems.

To understand this short tutorial you should already be familiar with SystemC. Good knowledge of C++ and object oriented design is a benefit but not necessary. You should also have some basic knowledge of hardware/software design.

1.1 Overview

OSSS is a SystemC based software library for high-level modelling of digital hardware/-software systems. As its name intends, OSSS has been designed with synthesis in mind. That means all modelling elements that can be used in an OSSS design have a well-defined synthesis semantic. A synthesis tool that integrates the synthesis concepts for OSSS is available, but not in the scope of this tutorial. For more detailed information about OSSS please refer to the OSSS manual [GBG+08] available from `http://www.system-synthesis.org`.

This tutorial is structured as follows:

Chapter 2 demonstrates how to set up a SystemC simulation environment. It guides through the installation process of the OSSS simulation library and delivers some insight into the internal organisation and design examples.

Chapter 3 gives a quick overview of the different abstraction layers used by the OSSS methodology: *Application Layer*, *Virtual Target Architecture Layer* and *Target Platform Layer*.

Chapter 4 demonstrates a top-down design of a packet processing application. It start with the definition of a simple producer/consumer design on the *Application Layer* using the following OSSS modelling elements: *Shared Object*, *Software Task*, and *Hardware Module*. In a separate step this initial HW/SW design is mapped onto different target architectures with different characteristics. This chapter closes with a discussion concerning the influence of the different mappings, in terms of measured packet throughput.

Chapter 5 provides contact information in case of technical questions regarding the public available OSSS simulation library. In contrast to the OSSS Library the *Fossy* synthesiser is not freely available. Readers with a further interest in using *Fossy* and trying synthesis of OSSS designs are welcome to contact Frank Oppenheimer `<frank.oppenheimer@offis.de>` by email to discuss the terms on which access to *Fossy* can be granted.

1.2 Typographical conventions

This manual uses the following typographical conventions. For continuous text the conventions shown in Table 1.1 are used. An example of a source code listing is shown in Listing 1.1. Comments are printed in italics while C++ keywords are printed in bold font. Special OSSS language elements are printed in blue color. To better emphasise certain parts of the source code printed in red color.

Convention	**Item**	**Example**
Times New Roman	Normal Text	This is an example sentence.
Times New Roman	Emphasised Text	This *word* is emphasised.
`Monospace font`	Class, function, method or macro names	`execute_operation()`

`Monospace italics`	Variables meant to be replaced when the language construct is used.
	Sometimes variables or parameters are omitted.
		`my_class<`*`parameter`*`>`
		`my_class<...>`
		`do_something(...)`

`Monospace font`	Shell commands	`make`

Table 1.1: Typographical conventions for continuous text

```
template<class ItemType, FIFO_size_t Size>
class FIFO : public FIFO_if<ItemType>
{
public:

  // Default constructor -- creates an empty FIFO
  FIFO();

  // Delete the contents of the FIFO
  OSSS_GUARDED_METHOD_VOID( Clear, OSSS_PARAMS(0), true );

  // Store an item in the FIFO if it is not full
  OSSS_GUARDED_METHOD_VOID( Put, OSSS_PARAMS(1, ItemType, item),
                            !OSSS_EXPORTED(isFull()) );

   [...]

private:

  // Increment index with wrap-around
  void IncrementIndex(FIFO_size_t &index);

   [...]

  // Buffer containing the items
  ItemType m_Buffer[Size];
};
```

Listing 1.1: Typographical conventions for listings

Listing 1.1 shows the typographical conventions for "terminal sessions". The > symbolises the user prompt. Like is the source code listings the [...] means that parts of the listing have been omitted.

```
> cd /
> sudo rm -rf *
[...]
```

Listing 1.2: Typographical conventions for "terminal sessions". But be careful trying this at home.

2 Getting Started

In this chapter we will describe the necessary steps to create a working environment for SystemC^{TM}, followed by a short example. So readers already familiar with SystemC can skip the following sections and continue reading at Section 2.3, where we will describe how to get started with OSSS.

2.1 Setting up a SystemC work environment

The first thing needed to work with SystemC is a C++ compiler, we assume such a compiler is already installed on your system, since Unix/Linux usually comes with a C++ compiler installed. Our recommendation is to use the GNU Compiler Collection [gcc] (gcc) in Version 3.4.4 (or higher). It is the most suitable compiler for the actual stable version of the SystemC library.

The second thing needed is the SystemC library which can be downloaded from [sysa]. To gain download access you have to register a user account (which is free of charge). At the time of writing, the actual stable version of the SystemC library is 2.2.0 (this version is compliant to the IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2005 [sys06]).

Download the “systemc-2.2.0.tgz” file to a local directory and extract the files contained within the archive. Change into the systemc-2.2.0 directory, set the environment variable “CXX” to the C++ compiler of your choice, e.g. “g++” on linux and execute “./configure”. Then compile and install the library by executing “make” and “make install”. If you get stuck compiling SystemC on your machine please consult the INSTALL file in the “systemc-2.2.0.tgz” file to get more detailed information.

Listing 2.1 shows the compilation and installation steps for a Linux machine using a tcsh C shell. A bash user only needs to change the `setenv CXX g++` command to `export CXX=g++`.

```
> tar -zxf systemc-2.2.0.tgz
> cd systemc-2.2.0
> mkdir objdir
> cd objdir
> setenv CXX g++
> ../configure --prefix=<desired/path/of/your/systemc/installation>
> make
> make install
> make check
> cd ..
> rm -rf objdir
```

Listing 2.1: Compiling SystemC 2.2.0 on a Linux machine using a tcsh C shell

2.2 Quick Introduction to SystemC

The SystemC package comes with a variety of examples you might want to have a look at. For a detailed introduction to SystemC we advise you to take a look at the SystemC User's Guide [sysa]. This guide also contains examples for designers that are familiar with VHDL or Verilog. For readers further interested in designing with SystemC books like [GLMS02] might also be of interest.

Basically, there are two different levels of abstraction for writing synthesisable models: register transfer level and behavioural level. When modelling on RTL, the designer defines the architecture of the design and the scheduling of the operations[1]. When modelling on behavioural level, the designer formulates the desired behaviour in an imperative way, similar to a sequential C program. The scheduling and the architecture are created by the synthesis tool.

2.2.1 Register transfer models

```
SC_MODULE(PointlessExample)
{
  sc_in<bool>  pi_bClk, pi_bReset, pi_bMode;
  // some ports of class type
  sc_in<Float<8,23> >                 pi_fData;
  sc_in< Matrix<Float<8,23>, 3, 1> > pi_Vector;
  sc_out<Matrix<Float<8,23>, 3, 1> > po_Vector;

  // determine the next state
  void NextState()
  {
    ms_bNextState.write(pi_bMode.read() == true &&
                        pi_fData.read() == 42);
  }

  // write the next state into the state register
  void UpdateState()
  {
    if (pi_bReset.read() == true)
      ms_bState.write(false);
    else
      ms_bState.write(ms_bNextState.read());
  }

  // determine output from state and inputs
  void Output()
```

[1]Depending on the actual modelling style there may be still some degrees of freedom concerning the architecture, allocation and binding. In this case these aspects are defined by the synthesis tool.

```
  {
     if (ms_bState.read() == true)
       po_Vector.write(pi_Vector.read());
     else
       // invoke a method on the object from the input
       po_Vector.write(pi_Vector.read().myMethod(pi_fData.read()));
  }

  sc_signal<bool>  ms_bState, ms_bNextState;

  SC_CTOR(PointlessExample)
  {
    // combinatorial processes
    SC_METHOD(NextState)
    sensitive << pi_bMode << pi_fData;
    SC_METHOD(Output);
    sensitive << ms_bState << pi_Vector << pi_fData;

    // sequential process with synchronous reset
    SC_METHOD(UpdateState);
    sensitive_pos << pi_bClk;
  }
};
```

Listing 2.2: An example of the basic structure of a synthesisable RTL OSSS module

Synthesisable RTL-SystemC models are `SC_MODULE`s that consist of `SC_METHOD`s, ports, internal signals and variables and further module instances. The `SC_METHOD`s can be combinatorial or sequential, i.e. synchronous to the clock and must not contain `wait()` statements. Resets can be synchronous or asynchronous depending on the sensitivity. An example is given in Listing 2.2.

2.2.2 Behaviour models

```
SC_MODULE(PointlessExample2)
{
  typedef Matrix<sc_fixed<16,1>, 4, 1> Vector_t;
  typedef Matrix<sc_fixed<16,1>, 4, 4> Matrix_t;

  sc_in<bool>         pi_bClk, pi_bReset;
  sc_in<bool>         pi_bDataAvail;
  sc_in<Vector_t>     pi_Vector;
  sc_out<Vector_t>    po_Vector;
  sc_out<bool>        po_bDataRdy;

  void Main()
  { // declare all local variables at the beginning (except
    // for for-loop indices
    Matrix_t Transform = Matrix_t::RotZ90;
    Vector_t result;

    // invoke method to reset all vector elements
    result.SetElements((Vector_t::BaseType)0);

    wait();
    while(true)
    {
      do { wait(); } while ( !(pi_bDataAvail == true) );
      po_Vector.write(pi_Vector.read()*Matrix);
      wait();
    }
  }

  SC_CTOR(PointlessExample2)
  {
    SC_CTHREAD(Main, pi_bClk.pos());
    reset_signal_is(pi_bReset, true);
  }
};
```

Listing 2.3: An example for a synthesisable behavioural level SystemC module

Synthesisable behavioural level SystemC models are `SC_MODULE`s that consist of `SC_CTHREAD`s, ports, internal signals and variables and further module instances. You cannot model purely combinatorial behaviour in `SC_CTHREAD`s due to the fact that they can be only sensitive to the specified clock edge. Resets are modelled by using the `reset_signal_is(...)` construct. An example is given in Listing 2.3.

2.3 Getting the OSSS Simulation Library

2.3.1 System Requirements

OSSS and OSSS+R are fully compliant to the IEEE Standard 1666-2005 [IEE05] for SystemC and can therefore be used with the OSCI reference implementation of SystemC in version 2.2.0. Previous versions of SystemC as well as third-party implementations with limited conformance to the standard may work, depending on the feature set used within the particular model.

- x86-GNU/Linux,
- OSCI SystemC 2.2.0
- GNU C++ compiler (gcc) version 3.4.2 or later, and
- GNU Make version 3.79 or later.

OSSS is developed with portability in mind. Hence, it can be easily ported to other platforms. It has been successfully tested on Sun SPARC Solaris and Cygwin, but in the future full support will only be provided for those platforms explicitly required and agreed on by the project partners.

2.3.2 Installation Instructions

For a successful build of the library and examples the installation script needs to know the locations of the SystemC include files and the SystemC library. The provided `Makefile` tries to guess these values automatically. However, if this fails one has to manually set the corresponding environment variables `SYSTEMC_HOME` and `SYSTEMC_LIB` to the correct locations, e.g.:

- For Bourne-shell compatible shells:
  `export SYSTEMC_HOME=/home/${USER}/systemc/systemc-2.2/`
  `export SYSTEMC_LIB=${SYSTEMC_HOME}/lib-linux-gcc`
- For C-shell compatible shells:
  `setenv SYSTEMC_HOME /home/${USER}/systemc/systemc-2.2/`
  `setenv SYSTEMC_LIB ${SYSTEMC_HOME}/lib-linux-gcc`

The OSSS simulation library can be downloaded from the OSSS & *Fossy* website [sysb] just follow the "Download" link in the main menu. Along with the actual library you will find an archive containing some examples and some other files. From this page you should download the library archive ("osss-2.2.0.tar.gz"). It includes the OSSS 2.2.0 simulation library and some examples that illustrate the use of the library. Some of these examples will be used during this manual as well.

After downloading the files you should extract the archive. Your directory tree should look as shown in Figure 2.1.

```
osss-2.2.0/
 ├─AUTHORS                          List of authors that have contributed to OSSS
 ├─COPYRIGHT                               Copyright preamble of each source file
 ├─LICENSE                                         The GNU lesser public license
 ├─Makefile                         Makefile for building the static part of the library
 ├─README               General information about the contents and the building process
 ├─build/
 │ Files that are needed for the building process
 ├─doc/
 │ Documentation (includes this manual)
 ├─examples/
 │ OSSS examples (see Figure 2.3)
 └─src/
   Source files (see Figure 2.2)
```

Figure 2.1: Structure of the OSSS 2.2.0 simulation library root directory

The OSSS-Library itself consists out of a header-only part and a part that needs to be compiled to a static library. Before you can start the compilation of the OSSS library make sure the SystemC library has been compiled and installed properly. Usually this is guaranteed when the `make check` from Listing 2.1 does not report any errors.

Now the OSSS simulation library can be built by typing `make` in the top level directory (this is the `simulation` directory as shown in Figure 2.1). Listing 2.4 shows the console output of a successful configuration of the OSSS simulation library. It is followed by the compilation of all cpp files and the creation of the `lib` directory and the `libosss.a` library.

```
osss-2.2.0> make
*** (Re-)creating OSSS library configuration...
Verbose compilation ...(default) no.
```

```
Looking for C++ compiler...(guessed) /opt/sw/tools/gcc/gcc-4.1.1/bin/g
   ++ (4.1.1).
Compilation flags...(default) -Wall.
Include debugging symbols ...(default) no.
Include profiling information ...(default) no.
Looking for SystemC headers ...
  '/opt/eda/systemc/systemc-2.2.0/include' (env)...yes.
Using SystemC with pthreads...(default) no.
Looking for SystemC library ...
  '/opt/eda/systemc/systemc-2.2.0/lib-linux_gcc-4.1.1' (env)...link...
     yes.
Configuration succeeded.
*** Building target 'all' (flavour: osss)...
[...]
```

Listing 2.4: Messages after successfully configuration of the OSSS library

```
src/
 └─o3s/
    ├─algorithms/
    │ Scheduling algorithms for Shared Objects and OSSS-Channels
    ├─communication/ Contains all communication related classes
    │  ├─app/
    │  │ Application Layer communication resource: OSSS-Port to a Shared Object
    │  └─arch/ Architecture Layer communication resources: simple point-to-point channel,
    │    simple RMI point-to-point channel, Xilinx OPB channel, and OSSS-Socket
    │     ├─osss_channel/
    │     │ OSSS Channel Generation/Service Layer
    │     └─osss_rmi/
    │       OSSS RMI Layer: OSSS-Port to RMI, OSSS Object Socket, OSSS RMI-Channel,
    │       and OSSS RMI Interface
    ├─datatypes/
    │ OSSS Array, OSSS Enumeration, OSSS Polymorphic Object, OSSS Serialisable Object,
    │ and OSSS Vector
    ├─extern/ Contains external sources
    │  ├─boost/
    │  │ Dedicated sources from the BOOST C++ library
    │  └─reflection/
    │    A macro based reflection mechanism for C++ [?]
    ├─structural/ Contains all structural related classes
    │  ├─app/
    │  │ Application Layer structural resources: OSSS Arbiter, OSSS Shared Object, and
    │  │ OSSS Software Task
    │  └─arch/ Virtual Target Architecture Layer class library of structural elements
    │     ├─interconnection/
    │     │ Xilinx OPB IP core wrapper
    │     ├─memory/
    │     │ OSSS Memory, Xilinx Block RAM, Xilinx external RAM
    │     ├─sw_processor/
    │     │ OSSS Processor, Xilinx MicroBlaze
    │     ├─system/
    │     │ OSSS System, Xilinx System, Xilinx external Port
    │     └─user_defined_hardware/
    │       OSSS Hardware Block, OSSS Module
    ├─utils/ Utilities used only inside of the library
    │  └─report/
    │    Error message, warning and text IO handling
    ├─osss top level include file that contains no imported namespaces (neither from SystemC
    │ nor OSSS)
    └─osss.h top level include file that contains all namespaces (from SystemC and OSSS)
```

Figure 2.2: Structure of the OSSS 2.2.0 simulation library `src` directory

For checking the correct configuration and installation of the OSSS simulation library it is recommended to compile the examples provided in the `example` directory. To compile all examples change into the “example” subdirectory and type `make`. The examples can be run by changing to a specific example subdirectory and typing `./run.x`. Figure 2.3 shows the structure of the OSSS 2.2.0 simulation library `examples` directory. A more detailed description of the examples provided with the OSSS 2.2.0 simulation library can be found in [GBG+08].

```
examples/
├─osss_channel_examples/ Examples that illustrate the use of OSSS-Channels
│ ├─i2c/
│ │ Example of an I2C bus
│ ├─simple/
│ │ Example of a simple point-to-point channel
│ ├─simple_bus/
│ │ Example of a simple bus with arbitration
│ └─simple_serialise/
│   Example of a simple but with arbitration and automatic datatype serialisation
├─osss_osss_examples/ All examples in this directory demonstrate the OSSS language
│ elements. All models are desribed on Application and Virtual Target Architecture Layer
│ ├─bridge/
│ │ Example of a bridge Shared Object between two different bus systems
│ ├─producer_consumer/
│ │ Producer/Consumer example using a FIFO Shared Object. We provide a simple and an
│ │ extended version with different target architecture mappings.
│ ├─raytracer/
│ │ Example of a Multi-CPU raytracer with a Shared Object for HW synchronisation and
│ │ image buffer
│ ├─shared_alu/
│ │ Example of an ALU (Arithmetic Logical Unit) implemented as Shared Object
│ ├─sw_timing/
│ │ Demonstration of execution time annotations and their effect on the simulation behaviour
│ └─transceiver/
│   Demonstration of a parallel to serial transceiver using Shared Objects
├─osss_recon_examples/ OSSS+R examples for demonstrating modelling dynamic recon-
│ figurable systems with OSSS
│ ├─audio/
│ │ Simple example of an audio decoding application supporting different codecs
│ ├─audio3/
│ │ Same as audio example above, but using a more elaborated modelling style
│ ├─car/
│ │ Simple car entertainment systems
│ ├─no_thrashing_audio/
│ │ Audio example that demonstrates the effects of thrashing due to fast configuration switches
│ ├─recon_sched_audio/
│ │ “no thrashing” audio example using a fair scheduler
│ └─two_process_audio/
│   Audio examples with two client processes accessing a single reconfigurable codec
└─osss_systemc_examples/ Pure SystemC examples, no OSSS specific language elements
  are used here
  ├─alarm_clock/
  │ Demonstrates the use of classes in a very simple alarm clock design
  └─video_filter/
    Demonstration of user classes with user-defined datatypes in a simple video filter example
```

Figure 2.3: Structure of the OSSS 2.2.0 simulation library `examples` directory

2.3.3 Troubleshooting

If there are any errors during the configuration phase of the build process, the user may resolve them by following the displayed suggestions. Alternatively, the user may enter `make help` to view additional build options. If the problem persists the OSSS developers can be contacted through the mailing lists (see Section 5.1).

3 The OSSS Methodology

The following sections describe how the OSSS methodology can be used to develop embedded systems. For simplicity we only present a top-down design flow (see Figure 3.1) that starts with a C++ "Golden Model".
In a first step, this entry model is decomposed into structural elements (called behaviours). This is done systematically by profiling trials that identify relevant and computationally intensive elements. After the identification and classification these structural elements are described by SystemC modules and OSSS *Software Tasks*. Communication and synchronisation between these parallel process elements is modelled by OSSS *Shared Objects*. They are special objects (instances of C++ classes) that provide a method interface for communication and guarantee a consistent access of an arbitrary number of concurrent processes. This first structured system description in OSSS is called *Application Layer Model* (AL Model). The application is described in SystemC and C++ under consistent involvement of object-oriented features that are supported by the OSSS methodology and the *Fossy* synthesis tool:

- Encapsulation of data and operations (methods) in classes. This is a basic object-oriented design principle for raising re-use.
- Method-based communication between structural blocks avoids effort of hand-crafted signal based communication and synchronisation. This kind of communication abstraction can be considered as high-level transaction level modelling (TLM), which is currently not covered by the OSCI TLM2 standard.
- Class inheritance allows easy extendibility for re-use. Polymorphism, which is based on inheritance relations, can be used to express implementation alternatives.
- Template classes offer easy parameterisation (e.g. buffer sizes) and make IP components more flexible.
- SystemC modules with ports and processes allow the specification of hardware components.
- OSSS Software Tasks specify parts of the application that are later executed on a processor.

On the AL we specify the function, logical structure, and an approximate time response of the system. Profiling results from analysis of the C++ "Golden Model" can be annotated to the AL to obtain an approximately timed behaviour. Also back-annotation approaches, where the execution of specific parts are profiled on their expected target technologies, are possible, but not further discussed here. Besides the functional correctness of the system, the

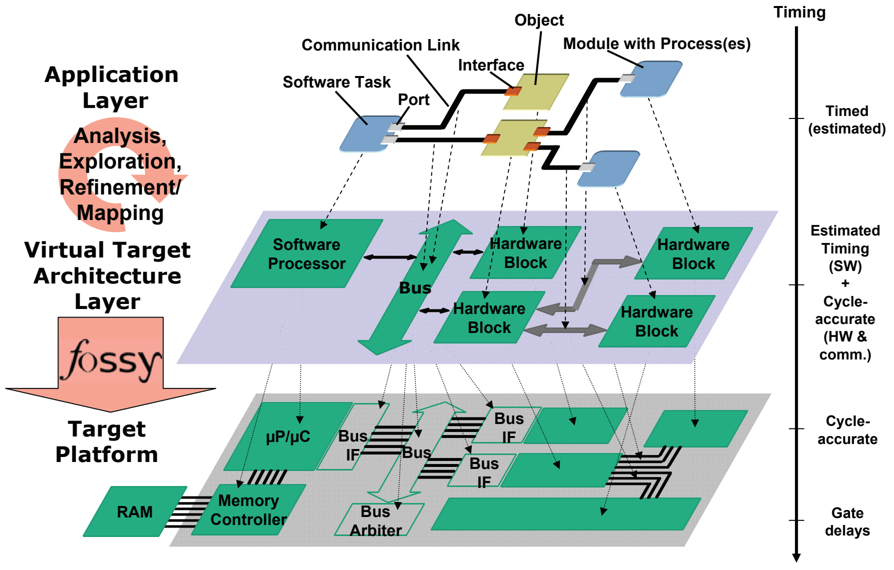


Figure 3.1: OSSS Design Methodology

AL offers an easy evaluation of design alternatives (e.g. HW/SW partitioning, scheduling, communication structures, and data locality). Profiling of different AL model alternatives with regard to their performance can be accomplished easily since the component's allocations and scheduling can be changed quickly. Analysis of the executable AL model in early design phases can help to detect and resolve bottlenecks in the logical structure. This might result in the relocation of timely critical computations from SW to HW or in reorganising complex computations in pipeline structures to enhance the throughput.

The next step towards the implementation is the refinement to a so-called *Virtual Target Architecture Model* (VTA Model). For this purpose more implementation details of the target architecture are added. Abstract communication relations from the AL model are mapped onto physical communication channels with different protocols and data bandwidths. By using different so-called OSSS-Channels [GBG+06] the designer can examine the influence of different bus protocols, data widths and arbitration schemes on the timing behaviour of the design. For saving costly communication resources, different logical communication relations can be grouped to a single physical connection. The resulting VTA model shows a cycle accurate timing behaviour and contains all other platform dependent information, like the chosen software processors or communication peripherals.

The VTA model is the input for the automatic *Fossy* synthesis process. It generates the overall system architecture, synthesisable VHDL for each hardware component, and C/C++ code for each software task. For the software parts a driver API and for the hardware parts a bus interface is automatically generated. Depending on the chosen platform, different so-called architecture description files can be generated. Special properties of different target platforms require adoptions of the synthesis process, e.g. for embedding special IP blocks or the generation of 3rd party tool specific configuration files. Figure 3.2 shows the *Fossy* synthesis process that has been tailored for Xilinx FPGA target technology.

In conclusion, the OSSS methodology for embedded HW/SW systems presents the following highlights:

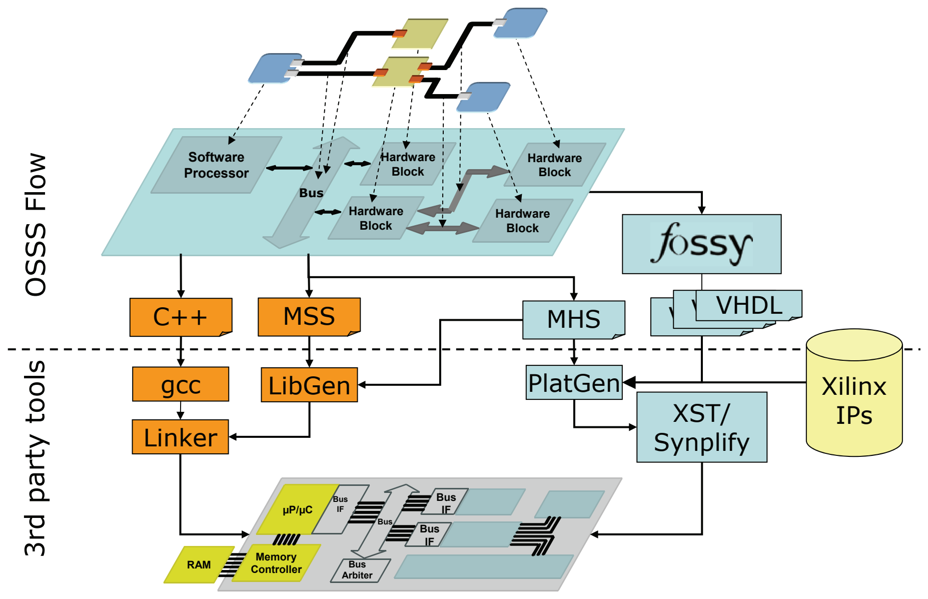


Figure 3.2: *Fossy* synthesis flow for Xilinx FPGAs

- Design and system synthesis of C++/SystemC 2.2 (ISO Standard) models in a homogeneous system-level description language and simulation environment.
- Stepwise and seamless refinement of hardware and software components.
- Easy integration of pre-existing IP components supported through encapsulation concept.
- Application Layer models allow the abstract modelling of the design and the exploration of different logical structures and schedules.
- Virtual Target Architecture models describe the physical properties and allow the comparison of alternative implementations on different platforms.
- Automatic synthesis with *Fossy* supports the OSCI SystemC Synthesisable Subset as well as manifold system-level modelling concepts (like Shared Objects, Software Tasks, and OSSS-Channels) for raising the designer's productivity.

4 Modelling HW/SW Systems with OSSS

4.1 OSSS Modelling Style

4.1.1 General

The modelling style in OSSS is very similar to the one of SystemC. This means that design structure is specified using modules, ports, and signals. The behaviour is described using (clocked) threads. However, OSSS allows to define and use objects within a synthesisable hardware design. Most of the basic concepts have already been introduced by OSSS targeting static hardware. OSSS+R adds new language elements for the specification of adaptive hardware systems (not convered in this tutorial, for further information please refer to [GBG+08]).

4.1.2 Method-based Communication

In (synthesisable) SystemC communication between modules and processes is modelled using ports and signals. OSSS introduced *Shared Objects* which provide method interfaces for the communication between processes and objects. This approach is quite similar to *Transaction Level Modelling* [RSPF05], but in difference to TLM there may be arbitrary methods which always keep a well-defined synthesis semantics. Hence, the designer does not have to manually refine those interfaces to signal-based interfaces.

4.1.3 Modelling Rule Checks

The library provides built-in checks for most of the design rules in OSSS. The checks are either tested at compile time, resulting in compiler errors on failure, or after the elaboration before the simulation starts. However, not all errors can be detected this way. For those cases, the designer has to take care of following the given design guidelines and rules.

4.1.4 Namespace Separation

The implementation of the OSSS modelling library uses C++ namespaces to separate public elements that can be used and accessed by a designer from internal parts that are implementation specific and must not be used outside the library. All public language elements are placed in the namespace `osss`, all internal elements are part of the nested namespace `osss::osssi`.

When specifying designs using the OSSS library the designer needs to include the OSSS header file either by `#include <osss.h>` or `#include<osss>`. In the first case, all OSSS language elements can be used without specifying their namespace, in the second case `osss::` has to be added as scope before all OSSS specific elements. It is recommended to use the second variant in order to avoid namespace pollution. However, for smaller designs and more convenience the user may also use the first one. In this report, for better readability, all given code examples use the first variant.

4.2 Application Layer

The Application Layer is the OSSS design entry for modelling hardware/software systems. On this layer the design methodology provides the concepts to model hardware modules, software tasks, passive inlined objects, and *Shared Objects* which can be shared between active parts of the design (see Figure 4.1).

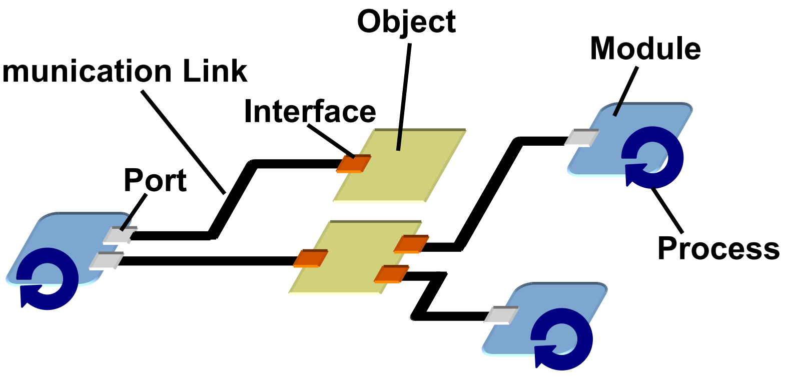


Figure 4.1: The *Application Layer* and its components

Active parts are hardware modules and software tasks that contain processes and thus have an own thread of execution. *Shared Objects* are passive which means they do not initiate any execution on their own. Shared Objects (see Section 4.2.1) have two major properties: on one hand they provide a method based interface for inter-process communication and on the other hand they are a kind of shared resource which can be used by different processes.

Figure 4.2 illustrates a simple producer/consumer design. The FIFO (First-In-First-Out) buffer between the producer and the consumer process is implemented using a *Shared Object*. This simple design will be used for the illustration of different OSSS modelling elements throughout this tutorial.

4.2.1 Shared Object

A new concept introduced by OSSS is the Shared Object `osss_shared<`*`userClass, schedulerClass`*`>`. While Shared Objects have no counterpart in C++, they can be used as a synthesisable replacement to model SystemC hierarchical channels, mutexes, semaphores, and events . They are called “shared” because their behaviour is shared between different

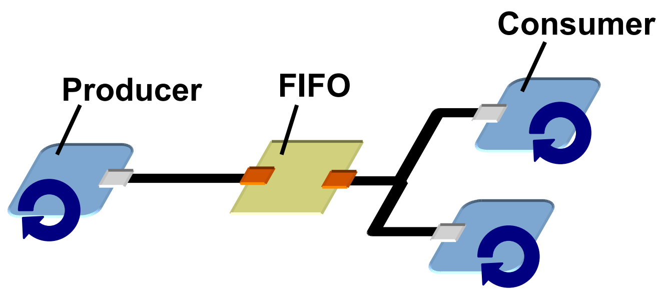


Figure 4.2: Producer/Consumer design on *Application Layer*

concurrent processes. Shared Objects are similar to the monitor concept used in some concurrent software description languages, because accesses always are mutually exclusive and arbitrated by a scheduler.

On the *Application Layer* an abstraction from the communication details is given. Communication initiated by either modules or software tasks with Shared Objects is performed by user-defined method calls through communication links. Argument and return types are of any valid C++ data type (except pointers & references). The communication link has the following properties:

directed: The source of a communication link is called a port while the destination is called an interface. Please note, that this does not necessarily mean that the data flow is from the port to the interface only. Since both methods with a void and a non-void return value can be called, the data flow is bidirectional.

blocking: A method call to a port does not return until the called method has been completed.

A Shared Object implements at least one interface and only reacts to external method calls on its implemented interfaces. These methods are called *guarded* because access to them is restricted by a so called *guard expression*. An access to such a method is only granted if this expression evaluates to true and the arbiter returns permission.

The behaviour of a Shared Object as well as the scheduling algorithm of the Shared Object's arbiter is custom-designed. Due to this flexibility Shared Objects can be used for a variety of different purposes:

Shared Objects can be used for a variety of different purposes:

- Interprocess communication and synchronisation.
- Method interface for modules.
- Modelling shared resources.

We call a process which contains a request to a method of a particular Shared Object a client or client process of that object. Likewise, we may also refer to a Shared Object as a server.

Using Shared Objects

To gain a better understanding of how Shared Objects are used in Application Layer models we are going to take a closer look at the consumer/producer design from the examples directory of the OSSS-Library. The idea of this design example is to use a Shared Object as a container for a user-defined FIFO class. The FIFO class provides a method interface for accessing a memory in a First-In-First-Out style. The Shared Object around the FIFO class enables resource sharing by providing arbitration facilities for multiple concurrent accesses. The structure of this example is shown in Figure 4.2.

Instantiation and binding of Shared Objects

Figure 4.3 shows a more detailed structure of the producer/consumer example on the *Application Layer*. A producer implemented as a software task calls methods defined in the `FIFO_put_if` on a local port, which is bound to a buffer Shared Object. Two hardware consumer processes call methods defined in the `FIFO_get_if`. The user-defined `FIFO` class inside the Shared Object container implements the put and get interface. The behaviour of the Shared Object instance is determined by the user-defined `FIFO` class. It is specified to store 10 items of type `Packet` A scheduling class (e.g. the pre-defined `osss_round_robin` or any other user-defined scheduling class) arbitrates concurrent accesses to the Shared Object containg the `FIFO`.
Listing 4.1 shows the OSSS Application Layer top-level design of the produer/consumer example as depicted in Figure 4.3. The communication links between the components are established by port to interface bindings: the output port of the producer (line 26) and both input ports of the consumer processes (line 33) are bound directly to the buffer Shared Object.
OSSS Shared Objects have two predefined ports. The `clock_port` and the `reset_port`. Both ports need to be bound to the global clock an reset signal of the design.

```
 1 #define OSSS_BLUE // Application Layer Model
 2 #include <osss.h>
 3 #include "Packet.hh"
 4 #include "FIFO.hh"
 5 #include "Producer.hh"
 6 #include "Consumer.hh"
 7
 8 SC_MODULE(Top) {
 9   sc_in<bool> clk, reset;
10
11   typedef osss_shared<FIFO<Packet, 10>,
12                       osss_round_robin> Buffer_t;
13
14   Producer  *m_Producer;
15   Buffer_t  *m_Buffer;
16   Consumer  *m_Consumer[2];
17
18   SC_CTOR(Top) {
19     m_Buffer = new Buffer_t("m_Buffer");
20     m_Buffer->clock_port(clk);
21     m_Buffer->reset_port(reset);
22
23     m_Producer = new Producer("m_Producer");
```

```
    m_Producer->clock_port(clk);
    m_Producer->reset_port(reset);
    m_Producer->output(*m_Buffer);

    m_Consumer[0] = new Consumer("m_Consumer0");
    m_Consumer[1] = new Consumer("m_Consumer1");
    for(unsigned int i=0; i<2; ++i) {
      m_Consumer[i]->clk(clk);
      m_Consumer[i]->reset(reset);
      m_Consumer[i]->input(*m_Buffer);
    }
  }
};
```

Listing 4.1: Top-Level module of the producer/consumer example on Application Layer

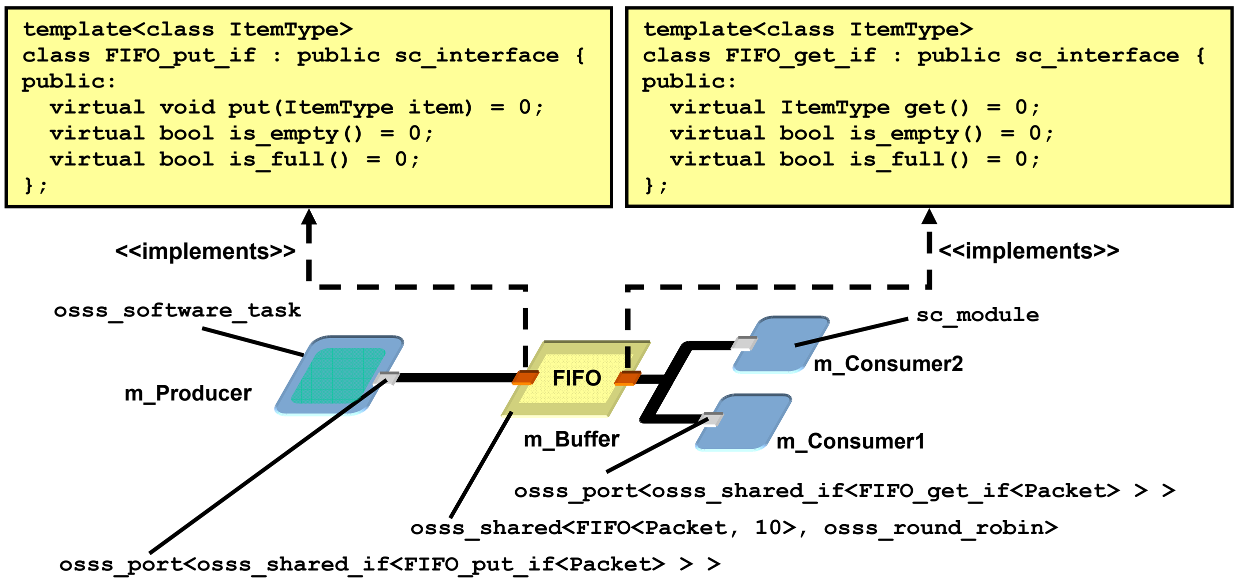

template<class ItemType>
class FIFO_put_if : public sc_interface {
public:
virtual void put(ItemType item) = 0;
virtual bool is_empty() = 0;
virtual bool is_full() = 0;
};
template<class ItemType>
class FIFO_get_if : public sc_interface {
public:
virtual ItemType get() = 0;
virtual bool is_empty() = 0;
virtual bool is_full() = 0;
};
<<implements>>
<<implements>>
osss_software_task
sc_module
m_Producer
FIFO
m_Consumer2
m_Buffer
m_Consumer1
osss_port<osss_shared_if<FIFO_get_if<Packet> > >
osss_shared<FIFO<Packet, 10>, osss_round_robin>
osss_port<osss_shared_if<FIFO_put_if<Packet> > >

Figure 4.3: Producer/consumer example on the *Application Layer*

Declaration of Shared Objects

In the following it is shown how a user-defined class, like the FIFO, has to be implemented to be usable as a Shared Object. First of all an abstract interface class needs to be specified. This abstract interface class specifies services the Shared Object provides for its attached client processes. It is possible to have more than a single interface per Shared Object.
Listing 4.2 shows two abstract interface classes (`FIFO_put_if` and `FIFO_get_if`), and the FIFO class implementation itself. The interfaces need to be defined separately from their implementation and need to be derived from the SystemC interface class `sc_interface`. Interfaces of Shared Objects are pure virtual, i.e. they consist out of pure virtual methods and do not contain any data members.
The FIFO class is derived from both interfaces, the `FIFO_put_if<...>` (line 20) and the `FIFO_get_if<...>` (line 21). Since the FIFO class is not allowed to contain any virtual methods it needs to implement all method derived from these interfaces. In a user class of

a Shared Object all exported (i.e. public accessible) methods are called *guarded methods*. All guarded methods are implemented using special `OSSS_GUARDED_METHOD` macros (e.g. line 30) to specify the method signature together with its associated guard condition (last argument). If no guard is specified the guard condition is constantly set to true (e.g. line 45). A guarded method used inside a guard condition needs to be wrapped by the `OSSS_EXPORTED` macro (e.g. line 31). The internal storage of the FIFO is described using an `osss_array` (line 60), which is a bounded vector of a user-defined type that can be converted easily to a physical memory (e.g. a Xilinx Block-RAM) during architecture refinement.

```
 1  typedef unsigned int FIFO_size_t;
 2
 3  template<class ItemType>
 4  class FIFO_put_if : public virtual sc_interface {
 5  public:
 6    virtual void put(ItemType item) = 0;
 7    virtual bool is_empty() = 0;
 8    virtual bool is_full() = 0;
 9  };
10
11  template<class ItemType>
12  class FIFO_get_if : public virtual sc_interface {
13  public:
14    virtual ItemType get() = 0;
15    virtual bool is_empty() = 0;
16    virtual bool is_full() = 0;
17  };
18
19  template<class ItemType, FIFO_size_t Size>
20  class FIFO : public FIFO_put_if<ItemType>,
21               public FIFO_get_if<ItemType>
22  {
23   public:
24
25    FIFO() : m_buffer(),
26             m_put_index(0),
27             m_get_index(0),
28             m_num_items(0) {}
29
30    OSSS_GUARDED_METHOD_VOID(put, OSSS_PARAMS(1, ItemType, item),
31                             !OSSS_EXPORTED(isFull()) ) {
32      m_buffer[m_put_index] = item;
33      increment_index(m_put_index);
34      m_num_items += 1;
35    }
36
37    OSSS_GUARDED_METHOD(ItemType, get, OSSS_PARAMS(0),
38                        !OSSS_EXPORTED(isEmpty()) ) {
39      ItemType result = m_buffer[m_get_index];
40      increment_index(m_get_index);
41      m_num_items -= 1;
42      return result;
43    }
44
```

```
  OSSS_GUARDED_METHOD(bool, is_empty, OSSS_PARAMS(0), true) {
   return m_num_items == 0;
  }

  OSSS_GUARDED_METHOD(bool, is_full, OSSS_PARAMS(0), true) {
    return m_num_items == Size;
  }

 protected:

  void increment_index(FIFO_size_t &index) {
    if (index == (Size-1)) index = 0;
    else index += 1;
  }

  osss_array<ItemType, Size> m_buffer;

  FIFO_size_t m_put_index, m_get_index, m_num_items;
};
```

Listing 4.2: FIFO interface and FIFO class implementation

Comparing the FIFO class from Listing 4.2 with a common C++ class implementation the main difference in the use of `OSSS_GUARDED_METHOD_VOID`, `OSSS_GUARDED_METHOD` and `OSSS_EXPORTED` constructs. These macros are provided by the OSSS library to bind a guard condition to a method. The guard condition determines whether the its associated method is accessible for client processes. In OSSS, the guard condition is only dependent from the inner state of a Shared Object. It is not possible that the guard condition is dependent on the parameters of the method.

In principle, any kind of user-defined C++ class can be used as a Shared Object. With the only restriction, that each method of such a class which should be callable by client processes needs to be declared in an abstract interface class and implemented as a guarded method.

A C++ method declaration with a `void` return type of the form:

```
void methodName(paramType1 param1, ...  paramTypeN paramN)
```

is translated to the following `OSSS_GUARDED_METHOD_VOID`:

```
 OSSS_GUARDED_METHOD_VOID( methodName,
                           OSSS_PARAMS(      N,
                                             paramType1, param1,
                                             ...
                                             paramTypeN, paramN),
                           guardCondition)
```

A C++ method with a non-`void` return type of the form:

```
return_type methodName(paramType1 param1, ...  paramTypeN paramN)
```

is translated to the following `OSSS_GUARDED_METHOD`:

```
OSSS_GUARDED_METHOD( return_type,
                     methodName, OSSS_PARAMS( N,
                                              paramType1, param1,
                                              ...
                                              paramTypeN, paramN),
                     guardCondition)
```

The main benefit of a Shared Object is that several clients can access the methods of that object without knowing about concurrent accesses from other clients. Thus, it is easy to add a client accessing a Shared Object without changing the Shared Object itslef. Furthermore, the guard conditions can be used to implement an implicit protocol; that is, to control the order of accesses for the inquiring clients.

Communication with Shared Objects

Communication with Shared Objects follows the SystemC IMC (Interface Method Call) mechanism. It consists of

Port-Interface Binding: For the establishment of a *Communication Link* a port of a module or software task needs to be bound to a Shared Object. This binding requires a port of the same type as the interface provided by the object. For calling methods on a Shared Object which implements the interface class `IF`, a port of type `osss_port<osss_shared_if<IF> >` needs to be bound.

Method Call: When the port is bound to a Shared Object is acts like a constant reference. Using the `operator->()` on the port allows calling each method which has been declared by the interface class. As mentioned before method calls to Shared Objects are blocking. They do not return control to the caller until the called method has been executed completely.

The schedulers

Concurrent accesses to guarded methods of a Shared Object are handled by a scheduler. The scheduling algorithm of a Shared Object can be changed easily by replacing the scheduler class. The OSSS-Library contains some pre-designed schedulers, these are listed in Table 4.1. Additional used-defined scheduling algorithms can be implemented easily.
For scheduling algorithms that support priorities these can be set by passing a positive number to the `setPriority()` method of an `osss_port<osss_shared_if<IF> >` during elaboration. The interpretation of this positive number (e.g. higher number means higher priority) is scheduling algorithm dependent.
Custom scheduling algorithms are implemented by deriving from class `osss_scheduler`. The derived class needs to implement the `PositiveNumber schedule( const RequestVector & clientRequests )` method and requires to be purely combinatorial, meaning it must not contain any `wait()` statement.

Scheduler	Description
`osss_round_robin`	• No priorities • Fairness not guaranteed
`osss_modified_round_robin`	• No priorities • Fairness not guaranteed, "fairer" than unmodified version
`osss_static_priority<`*ZeroIsHighestPrio*`>`	• Static priorities • Default: Zero is lowest priority • Fairness not guaranteed
`osss_ceiling_priority<`*MaxClients*`>`	• Dynamic priorities • Fair
`osss_least_recently_used<`*MaxClients*`>`	• Dynamic priorities • Fair

Table 4.1: Schedulers included in OSSS

Restrictions when using Shared Objects

To summarise the above the usage of Shared Objects is subject to the following restrictions.

- Shared Objects must implement a default constructor.
- All methods accessible from outside the Shared Object must be guarded.
- Direct access to data members is not possible.
- Shared Objects are passive and only react to requests from clients.
- Calls to guarded methods are blocking.
- Guarded methods are not allowed to be `const`.
- Parameters of guarded methods are not allowed to be of a pointer (`*`) or a reference (`&`) type.
- Parameters of guarded methods are not allowed to be `const`.
- Guard expressions must be free of side effects, they must not change the inner state of the Shared Object.
- Guards are only allowed to be dependant on the internal state of the Shared Object. I.e. the parameters of a guarded method are not allowed to be used in the associated guard evaluation.
- The evaluation of a guard expression must not cause the execution of a `wait()` statement.
- The guard expression must be satisfiable, meaning it must not be hardwired to false.

As for the Shared Object some restrictions apply to the clients that use Shared Objects:

- All client processes of a Shared Object must be driven by the same clock.
- Calls to methods of a Shared Object must be done by the `operator->()` method of the `osss_port<osss_shared_if<IF> >`. To complete this call a `wait()` statement has to follow after the `operator->()` call.
- Each `osss_port<osss_shared_if<IF> >` of a `sc_module` is allowed to be used by a single process only. If an `osss_port<...>` bound to a Shared Object is used by more than one process the simulation produces an error and is aborted immediately.
- Processes from which calls to Shared Objects originate must be `SC_CTHREAD`s.
- Parameters must be passed as values not references for the call to be synthesisable.

4.2.2 Software Task

Natively, SystemC does not support the modelling of software. Although it is very easy to write algorithms in a sequential and "untimed" or causal timed model, SystemC does not support a well defined synchronisation between hardware and software models. The simulation time is managed by the SystemC kernel and can only be advanced by calling the `wait()` function. The execution of the statements between two successive wait statements does not affect the simulation time maintained by the kernel. Hence, for a proper synchronisation of hardware and software components it is necessary to introduce a notion of software execution time.

One possibility to introduce execution times would be to use an explicit CPU model (e.g. based on an instruction set simulator) to execute the software. This approach has two main disadvantages. Firstly, the simulation performance of an instruction-level simulation is inferior to a native host code execution and secondly it complicates the introduction of an abstract communication mechanism between hardware and software. Therefore, we propose an approach based on the block-level annotation of execution times which overcomes these two disadvantages.

To overcome these limitations of SystemC a new class called `osss_software_task` is introduced. An OSSS Software Task is the counterpart to a `sc_module`. While an `sc_module` is a structural component specialised for the description of hardware, which is parallel by nature, and can contain an arbitrary number of (parallel) processes (`SC_METHOD`, `SC_CTHREAD` or `SC_THREAD`), an arbitrary number of `sc_modules` (hierarchical modules) and an arbitrary number of `sc_ports` for communicating with the "outside world", an `osss_software_task` is a structural component specialised for the description of sequential software. It only contains a single thread of control that is provided by a method called `main()` and implemented as `SC_CTHREAD`. For the `osss_software_task` two predefined ports, a `clock_port` and a `reset_port`, both of type `sc_in<bool>` are defined. These ports have to be bound to the top-level's global clock and reset signals. Beside these predefined ports, the software task can contain an arbitrary number of ports of type `osss_port<osss_shared_if<IF> >`. These ports are used to communicate with Shared Objects (see Section 4.2.1). In contrast to `sc_modules` no nesting of `osss_software_tasks` is allowed.

Before presenting the usage of software tasks by exmaple, Table 4.2 compares the properties of `sc_module` and `osss_software_task`.

	`sc_module`	`osss_software_task`
Purpose	Structural element for the modelling of parallel hardware	Structural element for the modelling of sequential software
Class declaration macro	`SC_MODULE(`*`class_name`*`)`	`OSSS_SW_TASK(`*`class`*`)` or `OSSS_SOFTWARE_TASK(..)`
Constructor macro	`SC_CTOR(`*`class_name`*`)`	`OSSS_SW_CTOR(`*`class`*`)` or `OSSS_SOFTWARE_CTOR(..)`
Number of processes	0 - N	1 (single thread of control)
Type of processes/ Notion of time	`SC_METHOD / next_trigger()`	-
	`SC_THREAD / wait(`*`time`*`)`	-
	`SC_CTHREAD / wait()`	`SC_CTHREAD / OSSS_EET(`*`time`*`)`
Special ports	-	`sc_in<bool> clock_port`
	-	`sc_in<bool> reset_port`
Communication ports	`sc_port<...>` (0 - N) `osss_port< osss_shared_if<...> >` (0 - N)	`osss_port< osss_shared_if<...> >` (0 - N)
	`sc_export<...>` (0 - N) `osss_export< osss_shared_if<...> >` (0 - N)	`osss_export< osss_shared_if<...> >` (0 - N)
	`sc_interface` (0 - N)	-
Nesting allowed	yes (arbitrary depth)	no

Table 4.2: Comparison between `sc_module` and `osss_software_task`

Declaration of a Software Task

Listing 4.3 shows the declaration of a software task in OSSS. For convenience the `OSSS_SOFTWARE_TASK` macro can be used instead of `class my_software_task : public osss::osss_software_task`. Similar to `SC_MODULE`s the default constructor can be written by using the `OSSS_SOFTWARE_CTOR` or `OSSS_SW_CTOR` macro. The method `main` is declared pure virtual in the base class `osss_software_task` and needs to be implemented by the user. This method represents the one and only thread of control per software task.

```
OSSS_SOFTWARE_TASK(my_software_task)
{

public:

  OSSS_SOFTWARE_CTOR(my_software_task)
  {
  }

  // alternative constructor
  my_software_task(...) : osss_software_task()
  {
    /* put your software task constructor
       code here */
    [...]
  }

  virtual void main()
  {
    /* put your software code here */
    [...]
  }
};
```

Listing 4.3: Declaration of a Software Task

Instantiation of a Software Task

Listing 4.4 shows the instantiation of `my_software_task` as declared in Listing 4.3. To "run" a software task, its pre-defined clock and reset ports need to be bound to the tob-level's clock and reset signals. Please note that both ports `clock_port` and `reset_port` are inherited from class `osss_software_task`. That is why these ports are available in class `my_software_task` and need to be bound, although non of them has been declard in Listing 4.3.

```
#define OSSS_BLUE
#include <osss.h>
#include "my_software_task.h"

SC_MODULE(Top)
{
public:

  sc_in<bool> clk;
  sc_in<bool> reset;

  my_software_task* mt;

  SC_CTOR(Top)
  {
    mt = new my_software_task("my_software_task");
    // perform binding of special clock and reset ports
    mt->clock_port(clk);
    mt->reset_port(reset);
  }
};
```

Listing 4.4: Instantiation of a Software Task

Using EETs for specifying the software timing behaviour

In our approach we distinguish two types of execution times: the **E**stimated **E**xecution **T**ime (EET) and the **R**equired **E**xecution **T**ime (RET). The EET as shown in Listing 4.5 defines the execution time of the enclosed code block. These annotated times will only affect the simulation and do not have any synthesis semantics. In principle these times can automatically be obtained (e.g. through profiling on the target CPU) and back-annotated into the model.

In order to achieve a realistic simulation it is necessary to impose two constraints on the usage of EETs. Firstly, no communication with other modules must happen within an EET block and, secondly, there must be no code between the end of an EET block and a communication statement. The EETs lead to a more accurate timing behaviour than relying on synchronisation through communication alone. By using an abstract communication mechanism with well defined points of interaction in conjunction with the two modelling constraints we are able to use timing annotations with block-level granularity without losing accuracy. When only using EETs, the accuracy of the overall simulation in terms of timing behaviour is determined by the accuracy of the defined EETs.

Listing 4.5 shows how to use `EET` blocks to specify the software timing behaviour. Besides the `OSSS_EET` macro we are using the `PRINT_MSG` macro which prints a kind of execution trace to the console. It is a kind of assertion which checks that the given time is "now". This macro does not influence the model execution, it just checks the correct behaviour of the timing annotations. Have a look at Listing 4.6 to see what this message macro writes to the console during the execution of `task1`. Each line in Listing 4.6 corresponds to a call of the `PINT_MSG` macro. The `EXPECTED_TIME` macro is used to check whether the proper simulation time has passed. When calling `EXPECTED_TIME(sc_time(110.0, SC_NS))` it is expected that exactly 110.0 nano seconds (ns) of simulation time has passed. If either more or less time has passed the macro writes an error to the console and quits the simulation. Like the `PRINT_MSG` macro the `EXPECTED_TIME` macro is some kind of assertion that do not influence the model execution either.

```
OSSS_SOFTWARE_TASK(task1)
```

```
{

public:

  OSSS_SOFTWARE_CTOR(task1)
  {
  }

  void methodX()
  {
    PRINT_MSG("Beginning methodX");
    OSSS_EET(sc_time(3.0, SC_US)) {
      /* do something else */
    }
    PRINT_MSG("Completed methodX");
  }

  virtual void main()
  {
    OSSS_EET(sc_time(2.0, SC_US)) {
      /* do something */
      PRINT_MSG("Doing something");
    }
    PRINT_MSG("Communication with some other module");

    EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                  sc_time(2.0, SC_US));

    // Note: The execution time of the initialisation of i and
    //       for checking the condition (at least the first
    //       time) is neglected here
    for (int i=0; i<3; ++i)
    {
      OSSS_EET(sc_time(5.0, SC_US)) {
        PRINT_MSG("For loop, iteration " << i);

        if (i%2 == 0)
        {
          // will be called for i==0 and i==2
          methodX();
        }
      }

      EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                    sc_time(2.0, SC_US)+
                    sc_time(5.0, SC_US)*static_cast<double>(i+1)+
                    sc_time(3.0, SC_US)*((i==2) ? 2.0 : 1.0)
                   );
    }
  }
};
```

Listing 4.5: Usage of EETs in a Software Task

Listing 4.6 shows the console output after ”running” the above software task. A clock period of 10.0 ns is used. Since the simulation starts with 10 synchronous reset cycles the first time stamp occurs at 110 ns.

```
110 ns    : top.task1(line: 94)   Doing something
2110 ns   : top.task1(line: 96)   Communication with some other module
2110 ns   : top.task1(line: 107)  For loop, iteration 0
2110 ns   : top.task1(line: 83)   Beginning methodX
5110 ns   : top.task1(line: 87)   Completed methodX
10110 ns  : top.task1(line: 107)  For loop, iteration 1
15110 ns  : top.task1(line: 107)  For loop, iteration 2
15110 ns  : top.task1(line: 83)   Beginning methodX
18110 ns  : top.task1(line: 87)   Completed methodX
```

Listing 4.6: Console output after running the Software Task from Listing 4.5 (Clock period is 10.0 ns, number of reset cycles is 10)

Using EETs and RETs for checking timing consistencies of Software Tasks

Syntactically the specification of RETs is almost identical to the specification of EETs, but the simulation semantics is different. The RET results in a piece of code which will not consume any simulation time at all. It can be considered as a timing constraint on the contained EET blocks and optional calls to the outside world (e.g. a Shared Object implemented in hardware).
It is also possible to mix and nest EETs and RETs. Doing so will allow for finding RET violations during the simulation. For instance, if an RET block of 5 ms contains a loop whose body has an EET of 1 ms per iteration and it performs more than 5 iterations in a simulation run, the RET block will report an error.
Listing 4.7 shows the usage of `EET` and `RET` blocks for checking timing consistencies of software tasks. In this example all `OSSS_EETs` are enclosed by `OSSS_RET` (**R**equired **E**xecution **T**ime) blocks. They report a timing violation when the amount of time that is passed inside an `RET` block is bigger than specified. In this example the `RET` in line 181 is intentionally violated by the inner `EET` block that is executed in a loop for three times. Please note that the timing violation is reported not until the `RET` block is left.

```
124 OSSS_SOFTWARE_TASK(task2)
125 {
126
127  public:
128
129   OSSS_SW_CTOR(task2)
130   {
131   }
132
133   void methodY()
134   {
135     OSSS_RET(sc_time(6.0, SC_US)) {
136
137       OSSS_EET(sc_time(4.0, SC_US)) {
138
139       }
140     }
141   }
```

```

  virtual void main()
  {
    PRINT_MSG("Beginning time critical calculation");
    OSSS_RET(sc_time(10.0, SC_US)) {

      OSSS_RET(sc_time(4.0, SC_US)) {

        PRINT_MSG("Beginning time critical sub-calculation 1");
        OSSS_EET(sc_time(2.0, SC_US)){
          /* do something */
        }
        PRINT_MSG("Completed time critical sub-calculation 1");

        EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                      sc_time(2.0, SC_US));

      }

      EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                    sc_time(2.0, SC_US));

      PRINT_MSG("Beginning time critical sub-calculation 2");
      OSSS_EET(sc_time(2.0, SC_US)) {
        /* do something */
      }
      PRINT_MSG("Completed time critical sub-calculation 2");

      EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                    sc_time(4.0, SC_US));

    }

    EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                  sc_time(4.0, SC_US));

    PRINT_MSG("Completed time critical calculation");

    PRINT_MSG("Beginning time critical calculation 2 (which will fail)");
    OSSS_RET(sc_time(3.0, SC_US)) {

      for (int i=0; i<3; ++i)
      {
        OSSS_EET(sc_time(2.0, SC_US)) {
          PRINT_MSG("For loop, iteration " << i);
        }
        EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                      sc_time(4.0, SC_US)+
                      sc_time(2.0, SC_US)*static_cast<double>(i+1));

      }
    }

    // The previous RET is intentionally violated by inner EETs.
    // Hence we expect now == 10.0 us instead of 7.0 us
    EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                      sc_time(10.0, SC_US));

    PRINT_MSG("Completed time critical calculation 2");

    PRINT_MSG("Beginning time critical calculation 3 (which is
        inconsistently constrained)");
    OSSS_RET(sc_time(5.0, SC_US)) {
      methodY();
    }

    // The previous RET is intentionally violated by an inner RET.
    // Hence we expect now == 10.0 us instead of 7.0 us
    EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
                      sc_time(14.0, SC_US));

    PRINT_MSG("Completed time critical calculation 3");
  }
};
```

Listing 4.7: Usage of EETs and RETs in a Software Task

Listing 4.8 shows the console output after "runnnig" the above software task. The `RET` violation in line 181 is reported as expected.

```
110 ns    : top.task2(line: 146)  Beginning time critical calculation
110 ns    : top.task2(line: 151)  Beginning time critical sub-calculation 1
2110 ns   : top.task2(line: 155)  Completed time critical sub-calculation 1
2110 ns   : top.task2(line: 165)  Beginning time critical sub-calculation 2
4110 ns   : top.task2(line: 169)  Completed time critical sub-calculation 2
4110 ns   : top.task2(line: 179)  Completed time critical calculation
4110 ns   : top.task2(line: 181)  Beginning time critical calculation 2
                                  (which will fail)
4110 ns   : top.task2(line: 187)  For loop, iteration 0
6110 ns   : top.task2(line: 187)  For loop, iteration 1
8110 ns   : top.task2(line: 187)  For loop, iteration 2
OSSS_RET violation [top.task2, sw_timing.cpp:182, created : 4110 ns] :
                    duration: 3 us, deadline: 7110 ns, now: 10110 ns
10110 ns : top.task2(line: 201)  Completed time critical calculation 2
10110 ns : top.task2(line: 203)  Beginning time critical calculation 3
                                  (which is inconsistently constrained)
14110 ns : top.task2(line: 213)  Completed time critical calculation 3
```

Listing 4.8: Console output after running the Software Task from Listing 4.7 (Clock period is 10.0 ns, number of reset cycles is 10)

4.2.3 Hardware/Software Communication

Listing 4.9 shows the producer to be implemented in software. To implement the producer as a software task the Producer class has to be derived from the `osss_software_task`. The communication of a software task with Shared Objects is performed by the usage of specialised OSSS-Ports. The `osss_port` is derived from the SystemC `sc_port` and is bound to the instance of the Shared Object, see Listing 4.1. The `osss_shared_if` class implements a Shared Object interface class used as a base class for the Shared Object class (`osss_shared`). Thus, the interface of the `osss_port` has to be of type `osss_shared_if` to connect the `osss_port` of the software task to a Shared Object. Furthermore, the interface of the object type of the Shared Object has to be specified as interface of the `osss_shared_if`. In Listing 4.9 the interface of the object type that is implemented as a Shared Object is `FIFO_if`. Thus the output port of the Producer class is of type `osss_port<osss_shared_if<FIFO_if> >`.

The `FIFO` in the producer/consumer example is specified to store items of type `Packet`. The implementation of the `FIFO` is explained in more detail in Section 4.2.1. The methods inside of the FIFO object are called from the software task by the `operator->()` on the `osss_port`, in the example the `put` method is called on the output port.

```
class Producer : public osss_software_task
{
public:

  // connection to the shared object
  osss_port<osss_shared_if< FIFO_if<Packet> > > output;

  // runs only once in the beginning
  OSSS_SW_CTOR(Producer) { }

  // has to override the virtual main()
  void main()
  {
    Packet p;
    while(true)
    {
      OSSS_EET(sc_time(50.0, SC_NS)) {
        /* some calculations that take approximately
           50.0 nano seconds */
      }

      // communication with the "outside world"
      output->put(p);

      OSSS_EET(sc_time(10.0, SC_NS)) {
        /* some calculations that take approximately
           10.0 nano seconds */
      }
    }
  }
};
```

Listing 4.9: OSSS-Software-Task with annotated Estimated Execution Times

Another software construct is the `OSSS_EET` (**E**stimated **E**xecution **T**ime) statement that serves as a notion of software execution time and is necessary for a proper synchronisation of hardware and software components.

Figure 4.4 shows how `OSSS_EET` statements are used in software tasks. The `OSSS_EET` statement defines the execution time of the enclosed code block on the target CPU. These annotated times will only affect the simulation and do not have any synthesis semantics. It is envisioned that these times can automatically be obtained and back-annotated from a tool.

In order to achieve a realistic simulation it is necessary to impose two constraints on the usage of EETs. Firstly, no communication with other modules must happen within an EET block and, secondly, there must be no code between the end of an EET block and a communication statement. The accuracy of the overall simulation in terms of timing behaviour is determined by the accuracy of the defined EETs. For further information regarding the `OSSS_EET` statement see Figure 4.4.

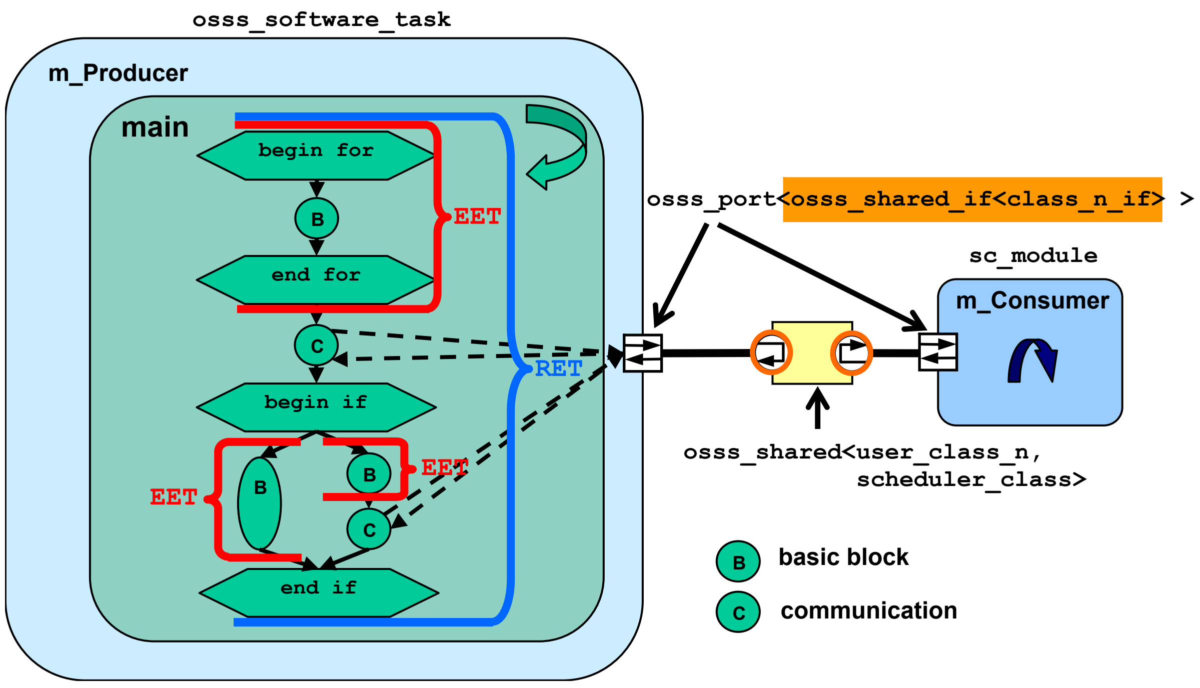


Figure 4.4: EET statements inside of the `main()` method of a software task

Listing 4.10 has the same block structure using EET and RET annotation as shown in Figure 4.4. The behaviour of this software task is to generate data of type `Packet` and to write them to a FIFO Shared Object. Until now we will only take a look on the block structure and the EET and RET annotations. The body of the infinite while loop (line 13) in the main process is constrained by an RET of 2000.0 nano seconds (line 14). The following for loop (line 18) initialises the `Packet` object and assigns a dummy payload. Since communication with Shared Objects can not be inside EET blocks the call of the `put` method on the `output` port (line 21) is not within the packet initialisation block. The same rule has been applied to the annotation of the following if condition (line 22); the else branch (line 28) contains a call to a Shared Object (line 34) and thus can not be enclosed by an EET block around the entire if condition.

```
1 OSSS_SW_TASK(Producer) {
2   osss_port<osss_shared_if<FIFO_put_if<Packet> > > output;
3
4   OSSS_SW_CTOR(Producer) : output("output") {}
5
6  protected:
  virtual void main() {
    const unsigned char source_addr = 42;
    unsigned char target_addr = 0;
    unsigned int offset = 0;
    Packet p;

    while(true) {
      OSSS_RET(sc_time(2000.0, SC_NS)) {
        OSSS_EET(sc_time(120.0, SC_NS)) {
          p.set_source_addr(source_addr);
          p.set_target_addr(target_addr);
          for(unsigned int i = 0; i<p.get_payload_size(); ++i)
            p.set_payload(i, i+offset);
        }
        output->put(p);
        if (target_addr >= 10) {
          OSSS_EET(sc_time(10.0, SC_NS)) {
            target_addr = 0;
            offset = 0;
          }
        }
        else {
          OSSS_EET(sc_time(30.0, SC_NS)) {
            target_addr += 1;
            offset += 10;
            p.set_target_addr(target_addr);
          }
          output->put(p);
        }
      }
    }
  }
};
```

Listing 4.10: Producer Software Task

Listing 4.11 shows the signature of the `Packet` class, which will be used in the following examples. It contains data members for a source and a target address and a payload of 10 bytes. To provide the concept of encapsulation the `Packet` class has several access methods to its protected data members.

```
class Packet {
 public:

  Packet();

  unsigned char get_source_addr() const;
  void          set_source_addr(unsigned char addr);

  unsigned char get_target_addr() const;
  void          set_target_addr(unsigned char addr);

  unsigned char get_payload(unsigned int index) const;
  void          set_payload(unsigned int index,
                            unsigned char data);
```

```

   unsigned int   get_payload_size() const;

  protected:
   unsigned char   m_source_addr;
   unsigned char   m_target_addr;
   unsigned char   m_payload[10];
};
```

Listing 4.11: Signature of the `Packet` class

4.2.4 Hardware Module

Hardware on the *Application Layer* is described by the OSSS hardware subset which is basically the synthesisable SystemC subset extended by C++ classes, inheritance and templates. A hardware module is an `SC_MODULE` with `SC_CTHREAD` and/or `SC_METHOD` processes which implement the behaviour. Ports are used to communicate with other components: SystemC signal ports are used to communicate directly with other hardware modules; OSSS ports are used to establish the communication with Shared Objects.

The consumer is an `sc_module` implementing a single clocked process, which calls the `get` method on its input port continuously (line 16). The `get` method called on the local port is redirected to a call of the guarded method implemented in the `FIFO` class, because the input port is bound to the buffer Shared Object. This abstract communication mechanism is uniform for SW Tasks and HW modules. It hides the details of the Shared Object's communication protocol involving the scheduling and guard evaluation.

```
 1 SC_MODULE(Consumer) {
 2    sc_in<bool> clk, reset;
 3 
 4    osss_port<osss_shared_if<FIFO_get_if<Packet> > > input;
 5 
 6    SC_CTOR(Consumer) : input("input") {
 7      SC_CTHREAD(cons_process, clk.pos());
 8      reset_signal_is(reset, true);
 9    }
10 
11  protected:
12   void cons_process() {
13      Packet p;
14      while(true) {
15        wait();
16        p = input->get();
17      }
18    }
19 };
```

Listing 4.12: Consumer HW module implementation

4.3 Virtual Target Architecture Layer

On this layer several architecture building blocks are available that can be used to assemble the overall system architecture. These building blocks are software processors, memories, and (user-defined) hardware blocks. For the interconnection of these blocks different communication channels, like buses, crossbar switches or point-to-point connections are available. All architecture building blocks are stored in a hierarchical *Architecture Class Library* that can be extended by user-defined architecture elements.

4.3.1 Architecture Class Library

Figure 4.5 shows the building blocks of the *Virtual Target Architecture* organised as a class hierarchy. The components shown are supported by the OSSS synthesis flow and can be used to build a synthesisable *Virtual Target Architecture*. The supported target architecture can be assembled from a subset of the Xilinx IP core library available in the Xilinx EDK (**E**mbedded **D**evelopment **K**it) and the Xilinx ISE (**I**ntegrated **S**ynthesis **E**nvironment) [**?**]. For the sake of simplicity Figure 4.5 only presents a set of selected architecture building blocks. Of course, the architecture class library can be easily extended by more Xilinx and custom components[1].

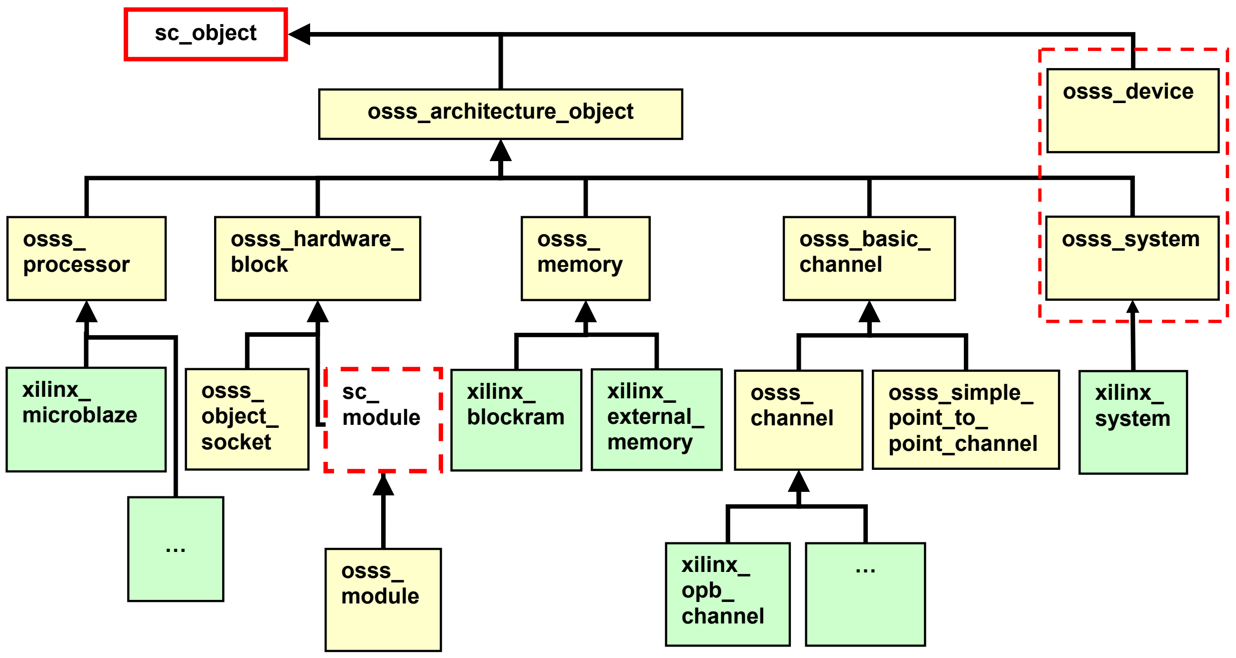


Figure 4.5: Sample of the OSSS Architecture Class Library

All architecture building blocks in Figure 4.5 with a `xilinx_` prefix are wrapper classes for configurable IP components provided by Xilinx. The other architecture building blocks are user-defined components: The `osss_hardware_block` is a base class for user-defined modules and OSSS Object Sockets. The `osss_module` is a specialisation of an `sc_module` and adds a mandatory clock and reset port to assure that all processes are driven by a global

[1]this is denoted by the blocks labelled with "...")

clock and reset signal. There is no semantical difference between the `sc_module` and the `osss_module`.
The OSSS-Channel is a concept to model the communication independently from the behaviour [GBG+06]. It can be used for a cycle accurate specification of a physical channel model, like an on-chip bus or a custom designed point-to-point channel. More details are given in [GBG+08].
The OSSS-Memory class is used for the explicit specification of memories in the Target Architecture. In Xilinx FPGAs these dedicated memories can be either internal Block-RAM or external memory, like SRAM, DRAM or Flash. The `osss_system` and the specialised `xilinx_system` are top-level modules which represent the system boundary. All ports of the `xilinx_system` are mapped to FPGA pins.

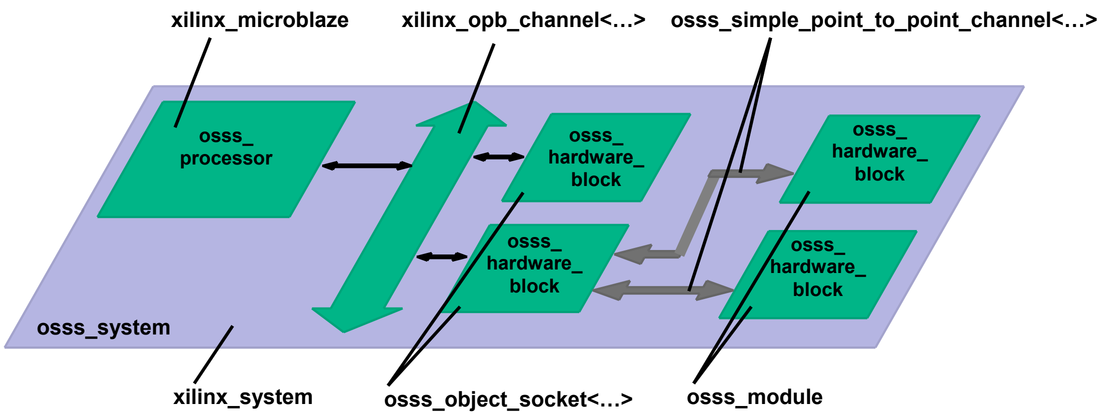


Figure 4.6: Example of a *Virtual Target Architecture* with a single processor and user-defined hardware

Figure 4.6 shows an example of a *Virtual Target Architecture* composed of different *OSSS Architecture Objects*. It includes a single Xilinx MicroBlaze^{TM}processor block connected to a Xilinx On-Chip Peripheral Bus (OPB) [IBM, Xil05] as bus master. Two *OSSS Object Sockets* are connected to the OPB as slave components. The lower OSSS Object Socket is connected with two user-defined hardware-blocks by a point-to-point connection.

4.3.2 Mapping the Consumer/Producer Design Example

In the following section we will map the simple consumer/producer example that has already been presented in Section 4.2 to the *Virtual Target Architecture Layer*. The upper part of Figure 4.7 shows the *Application Layer* consisting out of a producer (that is an `osss_software_task`), a Shared Object that contains a user-defined `FIFO` class (in this example is parameterised to hold up to 10 objects of type `Packet`) and a consumer (that is an `osss_module`). The lower part of Figure 4.7 shows the *Virtual Target Architecture Layer*, where the mapping and the communication refinement is accomplished.

In the following we will perform a stepwise communication refinement of the producer/-consumer example. To gain a better understanding about the amount of effort a designer has to spend, some code snippets will be presented. Whenever code from the *Application Layer* needs to be changed or extended we highlight these parts in red color.

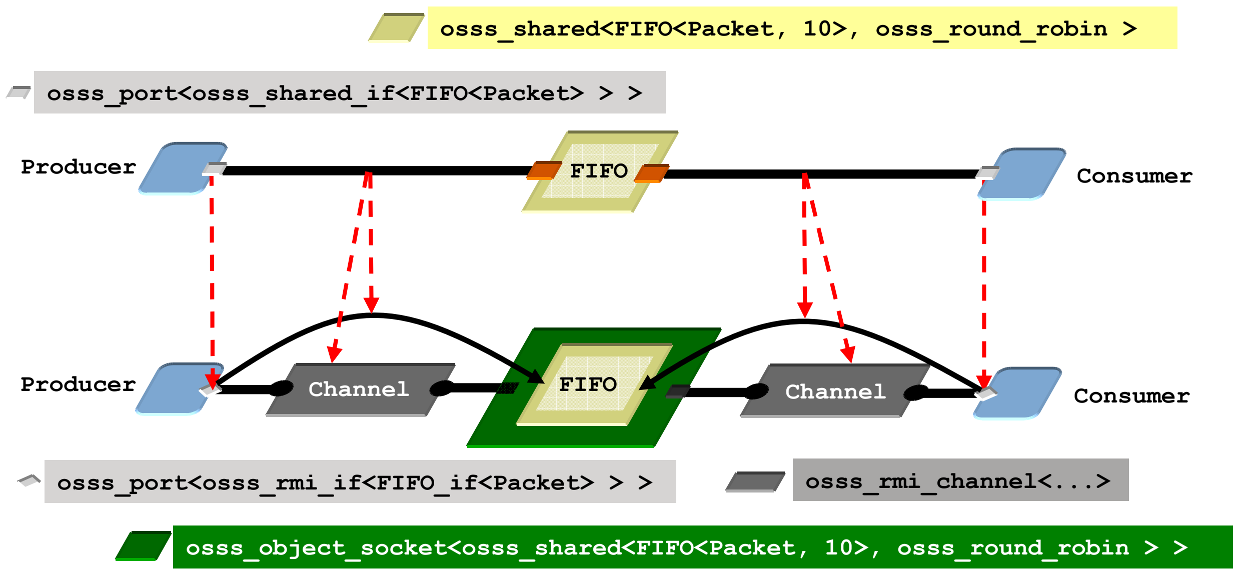

osss_shared<FIFO<Packet, 10>, osss_round_robin >
osss_port<osss_shared_if<FIFO<Packet> > >
Producer
FIFO
Consumer
Producer
Channel
FIFO
Channel
Consumer
osss_port<osss_rmi_if<FIFO_if<Packet> > >
osss_rmi_channel<...>
osss_object_socket<osss_shared<FIFO<Packet, 10>, osss_round_robin > >

Figure 4.7: Communication refinement of the producer/consumer example

As stated above, we assume that the hardware/software partitioning has already been performed (producer implemented as software, consumer implemented as hardware). Additionally, we assume that the behaviours of the producer, the FIFO, and the consumer are already specified in a synthesisable way. With these prerequisites the following refinement steps (the order is negligible) have to be performed:

1. Change all design elements of type `sc_module` to type `osss_module`.
2. Modify the ports of all Software Tasks and Hardware Modules. Change `osss_shared_if<`*IF*`>` to `osss_rmi_if<`*IF*`>`.
3. Build or generate `osss_rmi_if<...>` stubs for each Shared Object interface (for the producer/consumer example it is the `FIFO_put_if<...>` and `FIFO_get_if<...>`).
4. Equip all user-defined data types with serialisation support (in the producer/consumer example it is the `Packet` class).
5. Define custom OSSS-Channel or use pre-existing OSSS-Channel from Architecture Class Library (in the consumer/producer example the `xilinx_opb_channel` and the `osss_rmi_point_to_point_channel` are used).
6. Assemble the top-level design.

The `osss_rmi_if<...>` interface stub

All ports of the kind `osss_port<osss_shared_if<...> >` need to be replaced by ports capable of performing RMI (i.e. `osss_port<osss_rmi_if<...> >`). Listing 4.13 and Listing 4.14) show the replaced ports (changes regarding the *Application Layer* are marked in red color).

```
OSSS_SOFTWARE_TASK(Producer)
{
  osss_port<osss_rmi_if<FIFO__put_if<Packet> > > output;

  OSSS_SW_CTOR(Producer) { }

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

Listing 4.13: Producer with RMI capable port

```
OSSS_MODULE(Consumer)
{
  sc_in<bool>    pi_bClk;
  sc_in<bool>    pi_bReset;

  osss_port<osss_rmi_if<FIFO_get_if<Packet> > > input;

  sc_out<Packet>   po_Packet;

  SC_CTOR(Consumer)
  {
    SC_CTHREAD(main, pi_bClk.pos());
     reset_signal_is(pi_bReset, true);
  }

  void main();
};
```

Listing 4.14: Consumer with RMI capable port

Besides these port modifications, the designer has to provide an implementation of the `osss_rmi_if<...>` stub for each Shared Object interface. Concerning the other code of the producer and the consumer nothing has to be changed since they are already in a synthesisable state. Listing 4.15 shows the implementation of the RMI stub for the `FIFO_put_if<ItemType>` and `FIFO_get_if<ItemType>` interfaces.

The stub are derived from the `FIFO_put_if<ItemType>` and `FIFO_get_if<ItemType>` interface classes. This is necessary because we need to provide a dedicated stub for each method specified in the interface classes. The stub is created by the `OSSS_OBJECT_STUB_CTOR( _IF_type_ )` constructor, whereas `_IF_type_` is the type of the interface that needs to be implemented by this stub (It is usually the interface class the stub is derived from. In our example this are the `FIFO_put_if<ItemType>` and `FIFO_put_if<ItemType>` interfaces).

Hint: When using complex types inside macros it is usually a good idea make a `typedef` before (see lines 5 & 6 in Listing Listing 4.15). Consider the following example: Given a macro that takes a single argument like `#define MY_MACRO( _type_ )`, and type `My_Template_Type< a, b, c >;` using the macro `MY_MACRO( My_Template_Type< a, b, c > )` leads to a compilation error, since the pre-processor treats each comma as a separated argument. Using `typedef My_Template_Type< a, b, c > my_template_t` and instead `MY_MACRO( my_template_t )` leads to the desired behaviour.

Each method specified in the interface classes needs to be declared by either the `OSSS_METHOD_VOID_STUB(...)` or by the `OSSS_METHOD_STUB(...)` macro. The first macro is used for methods with a void return type while the second macro is used for methods with a non-void return type. These macros are very similar to the `OSSS_GUARDED_METHOD_VOID(...)` and the `OSSS_GUARDED_METHOD(...)` macros used inside the Shared Object's user-defined class implementation. The main difference is that the stub macros do not have a guard condition parameter.

```
template<class ItemType>
class osss_rmi_if<FIFO_put_if<ItemType> > : public FIFO_put_if<ItemType>
{
  public:
    typedef FIFO_put_if<ItemType> base_type;
    OSSS_OBJECT_STUB_CTOR(base_type);

    OSSS_METHOD_VOID_STUB(put, OSSS_PARAMS(1, ItemType, item));
    OSSS_METHOD_STUB(bool, is_empty, OSSS_PARAMS(0));
    OSSS_METHOD_STUB(bool, is_full, OSSS_PARAMS(0));
};

template<class ItemType>
class osss_rmi_if<FIFO_get_if<ItemType> > : public FIFO_get_if<ItemType>
{
  public:
    typedef FIFO_get_if<ItemType> base_type;
    OSSS_OBJECT_STUB_CTOR(base_type);

    OSSS_METHOD_STUB(ItemType, get, OSSS_PARAMS(0));
    OSSS_METHOD_STUB(bool, is_empty, OSSS_PARAMS(0));
    OSSS_METHOD_STUB(bool, is_full, OSSS_PARAMS(0));
};
```

Listing 4.15: RMI stubs for the `FIFO_put_if<ItemType>` and `FIFO_get_if<ItemType>` interfaces

The `osss_rmi_if<...>` acts as a stub or proxy to the remote object. Each method stub macro generates the appropriate code that is needed to perform a remote method invocation. This includes the determination of the method ID for the called method, the serialisation of all parameters, the transmission of this data through the bound RMI-Channel and the de-serialisation of the return parameter (for non-void methods only) received from the RMI-Channel.

Serialisation of user-defined data types

All data types that should be transferred via RMI have to be serialisable. That means each data type or user-defined class needs to be decomposable into chunks of a specific size in order to be transmittable through any channel of arbitrary data width. The OSSS library has support for all built-in C & C++ data types. Moreover, it supports all synthesisable SystemC data types. When dealing with user-defined data types like structs or classes some manual effort is required to make them serialisable.

Listing 4.16 shows the user-defined data type `Packet` that has been equipped with serialisation support. For making a user-defined class serualisable it needs to be derived from the class `osss_serialisable_object`. In addition, it needs use the `OSSS_IS_SERIALISABLE(_this_class_name_)` macro whose only argument is the type of the actual class.

```
class Packet : public osss_serialisable_object
{
public:
  OSSS_IS_SERIALISABLE(Packet);

  // default constructor
  OSSS_SERIALISABLE_CTOR(Packet, ());

  // copy constructor
  OSSS_SERIALISABLE_CTOR(Packet, (const Packet &pkt));

  // assignment operator
  void operator=(const Packet &pkt);

  // equality operator
  bool operator==(const Packet &pkt);

  virtual void serialise() {
    osss_serialisable_object::store_element(m_source_addr);
    osss_serialisable_object::store_element(m_target_addr);
    osss_serialisable_object::store_array(m_payload, 10);
  }

  virtual void deserialise() {
    osss_serialisable_object::restore_element(m_source_addr);
    osss_serialisable_object::restore_element(m_target_addr);
    osss_serialisable_object::restore_array(m_payload, 10);
  }

  unsigned char get_source_addr() const;
  void          set_source_addr(unsigned char addr);
  unsigned char get_target_addr() const;
  void          set_target_addr(unsigned char addr);
  unsigned char get_payload(unsigned int index) const;
  void          set_payload(unsigned int index,
                            unsigned char data);
  unsigned int  get_payload_size() const;

protected:
  unsigned char m_source_addr;
  unsigned char m_target_addr;
  unsigned char m_payload[10];
};
```

Listing 4.16: Adding de-/serialisation support to the user-defined `Packet` class

Each constructor has to be declared using the `OSSS_SERIALISABLE_CTOR( _this_class_name_, (_paramerter0_, ..., _parameterN_))` macro.

In the virtual methods `serialise()` and `deserialise()` all attributes of the actual class that need to be serialised/de-serialised have to be registered. The registration is performed by the `store_element(...)` and the `restore_element(...)` method for scalar types, and by the `store_array(...)` and the `restore_array(...)` method for array types. These store and restore methods are provided by the `osss_serialisable_object` base class. It is very important to notice that the sequence of the store methods calls in the serialise method needs to be exactly the same as the sequence of restore method calls in the deserialise method. Otherwise the resulting serialisation behaiour is undefined. This might become hard to debug, since the `serialise()` and the `deserialise()` methods are called "automatically" whenever a serialisation or de-serialisation action is required.

Other parts of user-defined data types are not affected.

The `osss_rmi_channel<...>` container for synthesisable OSSS-Channels

The `osss_rmi_channel<...>` is a container class for all OSSS-Channels which implement the `osss_abstract_channel` interface (e.g. buses or crossbar-switches). More simple OSSS-Channels which only implement the `osss_abstract_basic_channel` interface (e.g. point-to-point connections) need to be bidirectional in order to work inside an `osss_rmi_channel<...>` container.

```
typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
  HWSWChannelType;

typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
  HWHWChannelType;
```

Listing 4.17: Usage of the `osss_rmi_channel<...>` container

Listing 4.17 shows the usage of an `osss_rmi_channel<...>` container in the producer/-consumer example. The `HWSWChannelType` is a `xilinx_opb_channel<...>` with a least recently used scheduler and no registered grants. It allows the connection of multiple master and multiple slave components. Its data and address size is 32 bit. The `HWHWChannelType` is an `osss_rmi_point_to_point_channel<...>` that is a bidirectional point-to-point connection. Its data size is 8 bit in each direction.

The intended purpose of the `osss_rmi_channel<...>` is to separate the high-level RMI protocol from the low-level bit-accurate protocol of the channel. The channel protocol is implemented by the corresponding channel class (e.g. the `xilinx_opb_channel<...>` implements the protocol and manages the interconnection of the master and slave components as specified in the Xilinx specific implementation of the IBM On-Chip Peripheral Bus [IBM, Xil05]). All RMI protocol specific features that build on top of the channel protocol are implemented inside the `osss_rmi_channel<...>` class. The separation of RMI and the channel protocol makes it possible to design and test a channel independently from the more complex RMI protocol. The usage of well defined interfaces in both the `osss_rmi_channel<...>` and the OSSS-Channel allows to exchange one channel implementation by another. This substitution can be performed without any needs for modifying the rest of the design. This enables a convenient plug & play mechanism which allows easy exchange of physical channel implementations.

The `osss_object_socket<...>` container for Shared Objects

The `osss_object_socket<...>` container class for Shared Objects serves basically the same purpose as the `osss_rmi_channel<...>` container for OSSS-Channels. Firstly, it encapsulates the RMI protocol that is used for communication through channels (i.e. communication to the outside world) and secondly it encapsulates the method call performed on the Shared Object itself (i.e. internal communication, represents a virtual client calling a method on the Shared Objects inner class). This separation allows the designer to plug a Shared Object inside an `osss_object_socket<...>` container without the modifying any Shared Object code.

```
typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
                                       osss_round_robin> > BufferType;
```

Listing 4.18: Usage of the `osss_object_socket<...>` container

Listing 4.18 shows the usage of an `osss_object_socket<...>` that contains a Shared Object which contains a FIFO. Concurrent accesses are arbitrated by a round robin scheduling policy. When using this kind of object socket, the designer does not need to perform any code modifications, neither on the Shared Object nor on the `FIFO<...>` class inside of it.

The final assembly phase

In the final assembly phase we construct the top-level module containing the whole design mapped on the *Virtual Target Architecture Layer*. This involves the following steps:

1. Choose and **instantiate software processor(s)** available in the *Architecture Class Library*.
2. **Perform default mappings and substitutions**: map SW tasks to SW processors (single task per processor), substitute `sc_modules` by `osss_modules` and wrap Shared Objects by Object Socket containers.
3. **Instantiate RMI-Channel containers**. Choose OSSS-Channels from the *Architecture Class Library* or implement a synthesisable OSSS-Channel for your special needs. **Plug an OSSS-Channel into each RMI-Channel container**.
4. **Perform logical and physical bindings**: The *logical binding* represents the port to interface binding from the Application Layer Model. The *physical binding* describes the connection of the architecture building blocks (like processors, object sockets, hardware modules) to the RMI-Channel containers.

As one can see from these four steps, it does not require any changes of the behaviour of any software tasks or any hardware modules from the *Application Layer Model*. Nevertheless, after a profiling run of the Application Model mapped on the Virtual Target Architecture some changes or optimizations of the application's behaviour might become apparent. These changes can be performed separately on the *Application Layer Model* without affecting the chosen target architecture.

Listing 4.19 shows all modifications that have to be performed on the top-level design of the producer/consumer example. A graphical representation of the design described in

Listing 4.19 can be found in the lower part of Figure 4.7.

In the first part of Listing 4.19 two different kinds of RMI-Channels are defined (q.v. Listing 4.17). The `HWSWChannelType` is used for communication between the Producer (software) and the Shared Object (hardware). The `HWHWChannelType` is used for communication between the Consumer (hardware) and the Shared Object on the other side. This RMI-Channel definition is followed by the definition of the bounded Packet FIFO (`BufferType`) that is a Shared Object plugged into an `osss_object_socket<...>` (q.v. Listing 4.18).

```
#define OSSS_GREEN
#include "osss.h"

class Top : public osss_system
{
public:

  sc_in<bool>    pi_bClk;
  sc_in<bool>    pi_bReset;

  typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
    HWSWChannelType;
  typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
    HWHWChannelType;

  typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
                                          osss_round_robin> > BufferType;

  Producer*          m_Producer;
  HWSWChannelType*   m_Channel1;
  BufferType*        m_Buffer;
  HWHWChannelType*   m_Channel2;
  Consumer*          m_Consumer;

  xilinx_microblaze* m_Processor;

  sc_signal<Packet> ms_Packet;

  Top(sc_core::sc_module_name name) : osss_system(name)
  {
    m_Channel1 = new HWSWChannelType("m_Channel1");
    m_Channel1->clock_port(pi_bClk);
    m_Channel1->reset_port(pi_bReset);

    m_Channel2 = new HWHWChannelType("m_Channel2");
    m_Channel2->clock_port(pi_bClk);
    m_Channel2->reset_port(pi_bReset);

    m_Buffer = new BufferType();
    m_Buffer->clock_port(pi_bClk);
    m_Buffer->reset_port(pi_bReset);
    m_Buffer->bind(*m_Channel1);
    m_Buffer->bind(*m_Channel2);

    // this is a software task
    m_Producer = new Producer("m_Producer");
    m_Producer->clock_port(pi_bClk);
    m_Producer->reset_port(pi_bReset);
    m_Producer->output(*m_Buffer);

    m_Processor = new xilinx_microblaze("m_Processor");
    m_Processor->clock_port(pi_bClk);
    m_Processor->reset_port(pi_bReset);
    // this port binds the processor to its bus (m_Channel1)
    m_Processor->rmi_client_port(*m_Channel1);
    // here the above software task is added to this processor
    m_Processor->add_sw_task(m_Producer);

    m_Consumer = new Consumer("m_Consumer");
    m_Consumer->pi_bClk(pi_bClk);
    m_Consumer->pi_bReset(pi_bReset);
    m_Consumer->po_Packet(ms_Packet);
    m_Consumer->input(*m_Channel2, *m_Buffer);
  }
};
```

Listing 4.19: Modifications on the top-level module of the consumer/producer example

In the constructor of the top-level module `Top` both channels `m_Channel1` and `m_Channel2` are instantiated and bound to the global clock and the reset signal.

On the *Application Layer*, the producer and the consumer were both directly bound to the Shared Object. Due to the communication refinement, by inserting OSSS-Channels between the producer and the Shared Object as well as between the consumer and the Shared Object the bindings of the `m_Buffer`, `m_Producer` and `m_Consumer` need to be adapted. The producer and the Shared Object are bound to the same channel (`m_Producer` and `m_Buffer` are both bound to `m_Channel1`) and the consumer and the Shared Object are also bound to the same channel (`m_Consumer` and `m_Buffer` are both bound to `m_Channel2`).

When an `osss_object_socket<...>` is connected to a shared bus as a slave module an address map for that slave becomes necessary. An address map consists out of a base and high address that specify the address range a slave component is sensitive to. When a master drives the address lanes inside the OSSS-Channel the slave whose address range includes this address gets active and serves the masters request. Although it is possible to specify the address maps manually we suggest the designer not to do so unless he knows exactly what he is doing. In the normal case all address maps are calculated by the `osss_rmi_channel<...>` automatically.

When binding a port of type `osss_port<osss_rmi_if<...> >` to an `osss_rmi_channel<...>` on the *Virtual Target Architecture Layer* the binding information of the *Application Layer* stays intact. A second parameter of the `operator()` of the `osss_port<osss_rmi_if<...> >` class was introduced for that purpose. Having a look at the code in Listing 4.19 the output port of the producer is bound to the RMI-Channel and to the object that is plugged into the `osss_object_socket<...>` (i.e. the Shared Object itself). The need for retaining this binding information from the *Application Layer* is at least necessary for the simulation. Since, by the first method call performed on an `osss_port<osss_shared_if<...> >` the corresponding process doing this call is registered at the Shared Object. The same behaviour has to be retained after mapping the application to the *Virtual Target Architecture* and thus using `osss_port<osss_rmi_if<...> >`.

Until now OSSS is only capable of dealing with a single clock domain per system. This restriction can be exploited for writing more concise top-level modules. Listing 4.20 shows how to use the static `osss_global_port_registry` class to register the global clock and reset signal.

Each component of the *Virtual Target Architecture Layer* provides a clock and reset interface that defines a `clock_port` and a `reset_port`, both of type `sc_in<bool>`. The designer can either decide to perform a manual binding of these ports, or to omit the binding which results in an automatic binding to the globally registered clock and reset port.

When mapping a design from the *Application* to the *Virtual Target Architecture Layer* the designer has to take care to replace any `sc_module` by an `osss_module`. All other architecture building blocks like processors, object sockets, channels and memories are already capable of the automatic clock and reset port binding.

```
#define OSSS_GREEN
#include "osss.h"

class Top : public osss_system
{
public:

  sc_in<bool>    pi_bClk;
  sc_in<bool>    pi_bReset;

  typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
    HWSWChannelType;
  typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
    HWHWChannelType;

  typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
                                          osss_round_robin> > BufferType;

  Producer*          m_Producer;
  HWSWChannelType*   m_Channel1;
  BufferType*        m_Buffer;
  HWHWChannelType*   m_Channel2;
  Consumer*          m_Consumer;

  xilinx_microblaze* m_Processor;

  sc_signal<Packet> ms_Packet;

  Top(sc_core::sc_module_name name) : osss_system(name)
  {
    // register clock and reset ports and make them global
    osss_global_port_registry::register_clock_port(pi_bClk);
    osss_global_port_registry::register_reset_port(pi_bReset);

    m_Channel1 = new HWSWChannelType("m_Channel1");
    m_Channel2 = new HWHWChannelType("m_Channel2");

    m_Buffer = new BufferType();
    m_Buffer->bind(*m_Channel1);
    m_Buffer->bind(*m_Channel2);

    // this is a software task
    m_Producer = new Producer("m_Producer");
    m_Producer->output(*m_Buffer);

    m_Processor = new xilinx_microblaze("m_Processor");
    // this port binds the processor to its bus (m_Channel1)
    m_Processor->rmi_client_port(*m_Channel1);
    // adds the above software task to this processor
    m_Processor->add_sw_task(m_Producer);

    //CAUTION: Make shure the Consumer has been chagned from sc_module
    //         to osss_module. Otherwise automatic clock and reset port
    //         binding does not work!
    m_Consumer = new Consumer("m_Consumer");
    m_Consumer->po_Packet(ms_Packet);
    m_Consumer->input(*m_Channel2, *m_Buffer);
  }
};
```

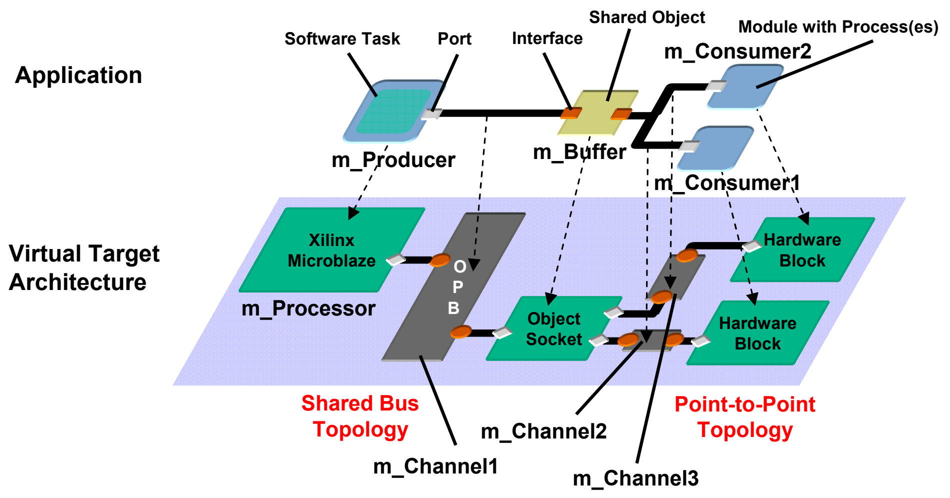
Listing 4.20: The top-level module from Listing 4.19 with global clock and reset port bindings

4.3.3 Architecture Exploration

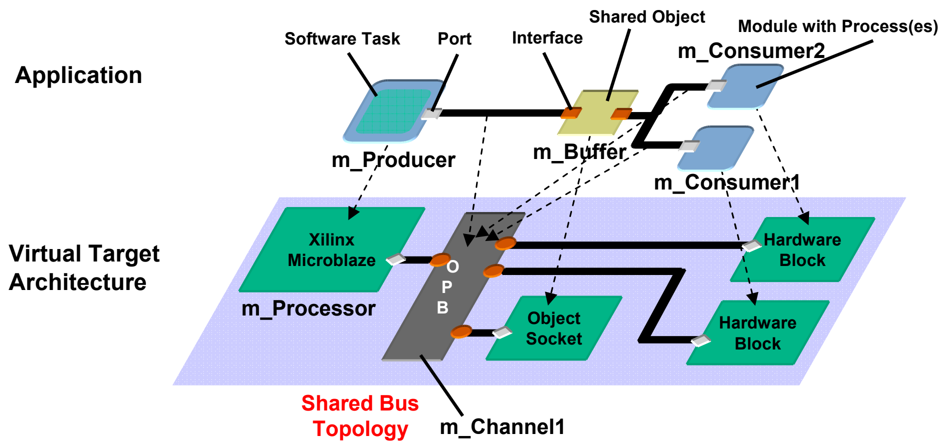
During the last section we have shown how to map an *Application* to a *Virtual Target Architecture Layer*. After presenting the basic mapping steps this section serves as a starting point for a simple top-down architecture exploration. During this section we present two different communication mappings of the producer/consumer example and discuss some of the profiling results generated from model execution. Moreover, the presented exmaple demonstrates the flexibility to quickly change the target architecture mapping.

In OSSS communication links (port to interface bindings) are mapped onto communication resources of the *Virtual Target Architecture Layer*, implemented as OSSS-Channels. The provided flexibility for the designer is very high, since these channels can differ in connection topologies (ranging from a point-to-point, over a shared bus to a full featured $N \times N$ crossbar-switch), bit sizes and in their communication protocols. Figure 4.8 illustrates different mapping alternatives of the producer/consumer example introduced in Figure 4.2.

In all mappings shown in Figure 4.8 the producer software task has been mapped to a Xilinx MicroBlaze$^{\text{TM}}$processor. It would have also been possible to map this task to any other processor available in the *Architecture Class Library*. One limitation of the current OSSS refinement methodology is that only a single SW Task can be mapped onto each



(a) Shared bus and point-to-point channel



(b) Single shared bus

Figure 4.8: Different mapping alternatives of the producer/consumer application

software processor. Currently we are working on the support of a lightweight operating system to support the mapping of N Software Tasks to a single processor.

Since the MicroBlaze processor has a built-in OPB interface we have connected it to a Xilinx OPB channel. However, it is also possible to connect any other channel to the MicroBlaze processor, but this would imply the use of a bride or protocol converter. For the sake of simplicity we have chosen the OPB in this example.

One of the main goals of the OSSS methodology is to provide a seamless synthesisable communication refinement for hardware/software systems. For the producer/consumer example this means the behaviours inside the producer software task and the consumer hardware modules are *not* affected by the mapping and communication refinement. In particular the high-level communication mechanism (of the *Application Layer*) using method calls on user-defined interfaces persists after mapping on the *Virtual Target Architecture Layer*.

The OSSS-Channels on the VTA Layer implement a synthesisable signal-level communication using architecture specific topologies and communication protocols, like the OPB. To retain the user-defined method calls from the application model we need a concept to translate them to this low-level communication resources. This kind of translation is usually performed by a network protocol stack defining several protocol layers that abstract from the underlying physical communication resource. In the OSSS methodology we call this concept **R**emote **M**ethod **I**nvocation (RMI). It enables the call of a method of a remote object through a physical (i.e. signal-level) connection. More details about the OSSS RMI concept will be given in [GBG+08].

Each communication link from the *Application Layer* can be mapped to any `osss_rmi_channel<...>` container (denoted by `m_Channel1 - 3` in Figure 4.8). They serve as wrappers for the OSSS-Channels, which implement the physical structure and the behaviour of communication protocols like buses (e.g. OPB) or point-to-point connections. The purpose of the RMI-Channels is the provision of a specific RMI interface and the translation of the OSSS-RMI protocol to the physical channel protocol.

Listing 4.21 shows the refined and mapped top-level design of the producer/consumer example on the Virtual Target Architecture Layer. The two different communication mapping alternatives from Figure 4.8 are marked with `ALTERNATIVE_A/B` pre-processor definitions. Another main difference between the Application and the Virtual Target Architecture Layer top-level design is the use of the `xilinx_system` base class (line 8). It serves two purposes: Firstly, it marks the top-level entity of the design used for synthesis. Secondly, it adds a “hook” to analyse the structure of the top-level design and generates target specific architecture definition and configuration files. When using the `xilinx_system`, configuration files for the Xilinx Platform Studio are generated during the SystemC elaboration phase.

At the beginning two different channel types are defined: the Xilinx OPB (`Bus_Ch_t`) and the point-to-point (`P2P_Ch_t`) channel with a client bit width of 8 (used by the initiator of the communication) and a server bit width of 32 (used by the target of the communication, i.e. Shared Objects). In the constructor (line 34) all channels are instantiated and bound to clock and reset. Depending on the mapping alternative the buffer Shared Object is bound to the physical connected channels using the `bind` method of the Object Socket (e.g. line 50). The producer SW Task is instantiated right before the Xilinx MicroBlaze which is bound to the OPB channel by its `rmi_client_port` (line 62). The `add_sw_task` method is used to map the producer SW Task to the MicroBlaze (line 63). After mapping the producer to the processor all communications using the `output` port of the SW Task are performed through the connected OPB channel. The binding of the `input` port of the consumer hardware modules gets a second parameter (line 71 & 72) on VTA. The first one defines the physical binding to a communication channel. The second parameter defines the same logical binding to the Shared Object as on the Application Layer.

```
1 #define OSSS_GREEN // Virtual Target Architecture Layer Model
2 #include <osss.h>
3 #include "Packet.hh"
4 #include "FIFO.hh"
5 #include "Producer.hh"
```

```
#include "Consumer.hh"

class Top : public xilinx_system {
 public:
  sc_in<bool> clk, reset;

  typedef
  osss_rmi_channel<xilinx_opb_channel<> >            Bus_Ch_t;

  typedef
  osss_rmi_channel<
    osss_point_to_point_channel<8, 32> >             P2P_Ch_t;

  typedef
  osss_object_socket<
   osss_shared<FIFO<Packet, 10>, osss_round_robin> > Buffer_t;

 protected:
  Bus_Ch_t *m_Channel1;
  P2P_Ch_t *m_Channel[2];

  Producer *m_Producer;
  Buffer_t *m_Buffer;
  Consumer *m_Consumer[2];

  xilinx_microblaze *m_processor;

 public:
  Top(sc_module_name name) : xilinx_system(name) {
    m_Channel1 = new Bus_Ch_t("m_Channel1");
    m_Channel1->clock_port(clk);
    m_Channel1->reset_port(reset);
#ifdef ALTERNATIVE_A
    m_Channel[0] = new P2P_Ch_t("m_Channel2");
    m_Channel[1] = new P2P_Ch_t("m_Channel3");
    for(unsigned int i=0; i<2; ++i) {
      m_Channel[i]->clock_port(clk);
      m_Channel[i]->reset_port(reset);
    }
#endif

    m_Buffer = new Buffer_t("m_Buffer");
    m_Buffer->clock_port(clk);
    m_Buffer->reset_port(reset);
    m_Buffer->bind(*m_Channel1);
#ifdef ALTERNATIVE_A
    m_Buffer->bind(*m_Channel[0]);
    m_Buffer->bind(*m_Channel[1]);
#endif

    m_Producer = new Producer("m_Producer");
    m_Producer->output(*m_Buffer);

    m_Processor = new xilinx_microblaze("m_Processor");
    m_Processor->clock_port(clk);
    m_Processor->reset_port(reset);
    m_Processor->rmi_client_port(*m_Channel1);
    m_Processor->add_sw_task(m_Producer);

    m_Consumer[0] = new Consumer("m_Consumer0");
    m_Consumer[1] = new Consumer("m_Consumer1");
    for(unsigned int i=0; i<2; ++i) {
      m_Consumer[i]->clock_port(clk);
      m_Consumer[i]->reset_port(reset);
#ifdef ALTERNATIVE_A
      m_Consumer[i]->input(*m_Channel[i], *m_Buffer);
#else // ALTERNATIVE_B
      m_Consumer[i]->input(*m_Channel1, *m_Buffer);
#endif
    }
  }
};
```

Listing 4.21: Top-Level module of the producer/consumer example on the VTA Layer

To demonstrate the impact of different communication mappings for the producer/consumer example we have performed a packet throughput measurement. The measurement has been performed on the Application Layer Model (ref. Figure 4.2) and the two different Virtual Architecture Models (ref. Figure 4.8). Table 4.3 shows the results of a simulation with 2000 produced packets at a clock frequency of 100.0 MHz. The simulation time is the duration of the entire simulation run measured on a reference workstation.

Table 4.3: Simulation results of the different producer/consumer models

Implementation Model	Simulation Time[a]	Packet Throughput
	[s]	[Packets/s]
Application Layer	0.2	12 512 512.5
Virtual Target Architecture Layer `ALTERNATIVE_A`: OPB & P2P channels	6.4	1 853 705.6
Virtual Target Architecture Layer `ALTERNATIVE_B`: OPB channel only	5.8	848 413.9

[a] Intel(R) Pentium(R) 4 CPU 3.00GHz

The *Application Layer Model*'s simulation time is the shortest while the measured packet throughput is the most highest. This result is not surprising since this layer abstracts from all communication details. The simulation runs of both *Virtual Target Architecture Models* take much longer (about a factor of 30) because they perform a cycle accurate simulation of the physical communication. A more interesting result is the significant lower packet throughput of mapping alternative B compared to mapping alternative A. Since alternative B uses only a single OPB channel we observe lots of bus contention that slows down the packet throughput dramatically.

5 Summary and Support

5.1 Support

To provide support for the users of the OSSS library OFFIS maintains several mailing lists. The subscription to *public* mailing lists can be initiated by sending a mail containing `subscribe <list-name>` in the body to `mdom@offis2.offis.uni-oldenburg.de`. The `<listname>` is the local part of the mailing list's address below.

- `osss-devel@offis.de` is a closed mailing list which can be used to contact the developers of the OSSS library. Subscription requires approval of the list maintainers, mails to this list can be sent from any e-mail address.
- `osss-user@offis.de` is a public mailing list for discussions on the usage of the above-mentioned libraries. Sending mails to the list requires prior subscription. Announcements of major changes and other important news are sent to this list as well.

5.2 Conclusion

The basic features and modelling techniques of OSSS, have been introduced by a simple procuder/consumer design. More information can be found within the OSSS source code package. The presented library targets flexible hardware/software communication for different target architectures. More advanced features and details concerning synthesis can be found in [GBG+08].

References and Further Reading

[GBG⁺06] Kim Grüttner, Claus Brunzema, Cornelia Grabbe, Thorsten Schubert, and Frank Oppenheimer. OSSS-Channels: Modelling and Synthesis of Communication With SystemC. In *Proceedings: Forum on Specification & Design Languages*, September 2006. 16, 41

[GBG⁺08] Kim Grüttner, Claus Brunzema, Cornelia Grabbe, Philipp A. Hartmann, Andreas Herrholz, Henning Kleen, Frank Oppenheimer, Andreas Schallenberg, and Schubert Thorsten. *OSSS - A Library for Synthesisable System Level Models in SystemC^{TM}, The OSSS 2.2.0 Manual*, 2008. 5, 14, 19, 41, 53, 57

[gcc] Homepage of the GNU Compiler Collection (gcc) Project. `http://gcc.gnu.org/`. 9

[GLMS02] Thorsten Groetker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002. 10

[IBM] IBM. *On-Chip Peripheral Bus Architecture Specifications, Version 2.1, SA-14-2528-02*. 41, 46

[IEE05] IEEE Standards Association ("IEEE-SA") Standards Board. *IEEE Std 1666-2005 Open SystemC Language Reference Manual*, 2005. 12

[RSPF05] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction Level Modeling in SystemC. Technical report, Mentor Graphics; Cadence Design Systems, 2005. 19

[sysa] Homepage of the Open SystemC Initiative. `http://www.systemc.org`. 9, 10

[sysb] Homepage of the OSSS and *Fossy* Project. `http://www.system-synthesis.org`. 13

[sys06] *IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2005*, March 2006. 9

[Xil05] Xilinx. *On-Chip Peripheral Bus V2.0 with OPB Arbiter (v1.10c), DS401*, June 2005. 41, 46