
R&D Division Transportation
Hardware/Software Design Methodology Group


# OSSS- A Library for Synthesisable System Level Models in SystemC™

## The OSSS 2.2.0 Manual

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: March 12, 2009

# Contents

# 1 Introduction

## 1.1 To whom this manual is directed?

This manual is for people who want to learn about OSSS and the OSSS synthesis flow. Since OSSS is an extension of SystemC™, profound knowledge of the SystemC language or other HDLs like VHDL or Verilog is useful, although not essential for the understanding of this manual.

## 1.2 Structure of this manual

This manual is divided into eight parts, some of these, especially Chapter 8, may not be of interest to every reader as they address the usage of the *Fossy* synthesis tool, which is not publicly available.

Chapter 2 presents the basic ideas of OSSS and motivates why a designer should use the OSSS methodology. The last part of this chapter explains how OSSS extends SystemC and gives an overview of the basic language concepts introduced by OSSS.

Chapter 3 is mostly intended for designers that are new to SystemC and OSSS and want to get started. It explains how to obtain and install both the SystemC and the OSSS simulation library.

Chapter 4 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 5 summarises the OSSS modelling style in terms of method-based communication, synthesis subset requirements, modelling rule checks, and namespace separation.

Chapter 6 presents the OSSS modelling elements of each abstraction layer. For the *Application Layer* the modelling elements are subdivided into software and hardware related parts. The software related part introduces *OSSS Software Tasks* and the software timing model. The hardware related part introduces *OSSS Shared Objects*, that can be used for abstract hardware/software communication. Moreover, we introduce *OSSS Named Contexts* and *OSSS Recon Objects* for the description of run-time adaptive hardware.
The description of the *Virtual Target Architecture Layer* starts with a classification of its basic building blocks into a so-called *Architecture Class Library*. This is extended by special device descriptor for the use of run-time reconfigurable hardware (e.g. Xilinx Virtex-2/4/5

FPGAs).
An overview of the OSSS mapping and refinement process from Application to Virtual Target Architecture is given. This includes the OSSS RMI (Remote Method Invocation) technology, one of the key enablers for adaptive communication mapping and refinement in OSSS. OSSS-Channels are a special part of the Architecture Class Library. An excursus to this concept is given, since it enables the designer to build synthesisable user-defined physical channels with user-defined protocols. In particular we introduces a simple point-to-point channel and a channel with arbitration (shared bus).
All mapping steps are illustrated using a simple producer/consumer example. Finally, we present how the OSSS methodology can be used to support communication architecture exploration.

Chapter 7 introduces two design patterns that are based on the OSSS modelling elements from the previous chapter. The *Mediator Pattern* utilises an OSSS Shared Object to implement hardware/software communication with dynamic run-time reconfigurable hardware. The *Prefetching Pattern* describes an optimisation strategy for dynamic run-time reconfigurable hardware. It hides reconfiguration times by configuration prefetching.

Chapter 8 covers the synthesis process. It first explains the architectural context extraction from the *Virtual Target Architecture Layer* followed by the hardware/software architecture sysnthesis technology. Afterwards this chapter is split up into the software library synthesis and the high-level hardware synthesis (performed by *Fossy*), both working hand in hand for the hardware/software interface synthesis. This chapter closes with a section about the synthesis subset accepted by the current *Fossy* release.

Chapter 9 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.

## 1.3 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.<br>Sometimes variables or parameters are omitted. | `my_class<`*`parameter`*`>`<br>`my_class<...>`<br>`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 What is OSSS

The OSSS methodology has been developed in three successive IST research projects:

- ODETTE (**O**bject-oriented co-**DE**sign and functional **T**est **TE**chniques) [ODEa]
- ICODES (**I**nterface- and **CO**mmunication-based **D**esign of **E**mbedded **S**ystems) [ICOa]
- ANDRES (**AN**alysis and **D**esign of run-time **RE**configurable, heterogeneous **S**ystems) [AND]

It provides an efficient design flow for communication dominated hardware/software systems, and allows the designer to start with an abstract executable specification containing many communicating hardware and software components. Designs written in OSSS can be easily refined and automatically synthesised to various target architectures and technologies.

## 2.1 Why OSSS?

The OSSS (**O**ldenburg **S**ystem **S**ynthesis **S**ubset) modelling library is based on SystemC™ [IEE06]. This library enables the combination of object-oriented concepts from software engineering with digital hardware design. While raising the level of abstraction by introducing object-orientation and transaction level modelling (TLM), it keeps well-defined synthesis semantics for all available modelling elements allowing automatic and efficient synthesis of OSSS models to RT-level VHDL or SystemC models. Those RT-level models can be further processed using well established RTL-to-Gate-level synthesis tools.

OSSS is mainly motivated by the use of the object-oriented features of C++ in a synthesisable SystemC model. As the term "class library" suggests, SystemC internally makes use of the object-oriented features of C++, e.g. classes, inheritance, polymorphism, communication by method calls, etc. However, these features are not available to the designer when writing synthesisable models. Synthesisable models (RTL and behavioural level) have to be constructed from modules with ports, connected via signals with behaviour described in processes. Even when neglecting synthesis, the use of these features in pure simulation models can be problematic[1]. This is where OSSS applies. By allowing many features for synthesis that are already part of SystemC and C++ and by adding new

[1]Consider, for example the case when two different processes simultaneously call methods of the same object, which may lead to an undefined state - similar to writing shared variables from different processes.

constructs, the level of abstraction and the expressiveness are increased.

Another challenge that arises during the integration of hardware and software is the hardware/software communication. When using SystemC to develop an executable specification of a hardware/software design on a functional level, it is impossible to synthesise it for a certain hardware platform without extensive manual refinement. Especially the implementation of the hardware/software communication is a very error prone and time consuming task. In order to avoid this manual refinement efford, or to keep it as low as possible the OSSS Methodology provides a strict separation of communication, synchronistaion and computation. Morevoer, this separation is supported by two abstraction layers. The initial design is described on the so-called *Application Layer*. By using well defined mapping rules the design is refined to the *Virtual Target Architecture Layer* that enriches the pure application with architecture information (e.g. which kind of CPUs are used for the execution of the software and which kind of channels or buses are used for hardware/software communication). Finally, the mapped application can be synthesised for a specific target platform. Up to now only a synthesis back-end for the Xilinx FPGAs exists but it is envisioned to support more target platforms in the future.

## 2.2 Shared Objects

Communication modelling is one of the key aspects in the design of complex embedded systems. Methodologies and techniques that help to improve the modelling of communication between subcomponents can also significantly improve the whole design process. For this reason SystemC provides the concept of channels for communication. A channel can be used to abstract from the details of a certain communication - for instance, a certain protocol -, and allows processes to communicate and exchange data with each other, based on method calls instead of signals. This concept already has led to a new kind of design methodology, called transaction level - or transaction based - modelling [RSPF05, OSC].

Indeed the channel concept provides for modelling communication at a higher level, and to separate communication and behaviour more cleanly. But unfortunately, channels in general do not possess a clear synthesis semantics. And, although being intended for communication between concurrent components, they do not posses built-in mechanisms for handling concurrent accesses. This makes channels quite useful for creating untimed models and for simulation but not for synthesis.

For these reasons we offer an alternative concept for SystemC channels, based on so-called Shared Objects. Like any other object it may specify a set of methods that form its interface. This allows processes to communicate and exchange data via Shared Objects based on method calls, as illustrated in Figure 2.1, similar to SystemC channels. Like SystemC ports and signals, Shared Objects which are located in different modules can be bound to each other, thus enabling communication throughout a hierarchy of modules.

In contrast to channels, a Shared Object possesses built-in mechanisms for handling concurrent accesses, a clear synthesis semantics, and it exhibits a timed behaviour during simulation. Handling concurrent accesses is basically realised by means of a scheduler that can

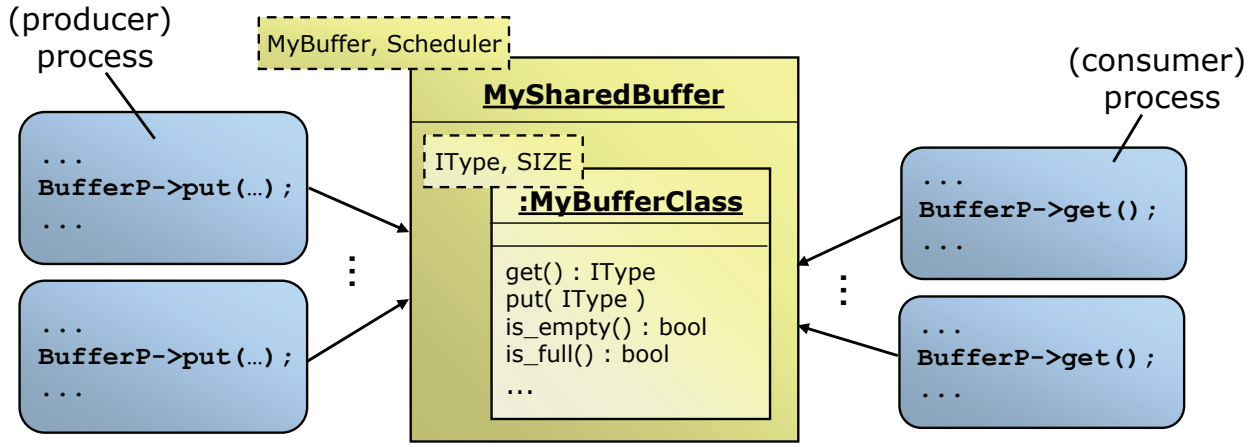



Figure 2.1: Accessing a Shared Object from multiple producer and consumer processes.

be specified by the user for each Shared Object. The scheduler determines which client process is granted to access the Shared Object in the case of concurrent requests. All other requesting clients are blocked meanwhile. As a consequence accesses are mutual exclusive. In addition this is supported by a guard mechanism which allows one to associate a Boolean expression - the so called guard - with a member function. Clients requesting an operation whose guard is false at that moment are ignored for scheduling and are blocked. The timed behaviour of Shared Objects give the designer an early and realistic impression on the temporal behaviour of the modelled system during simulation even before performing synthesis.

## 2.3 The OSSS+R extension

OSSS+R is a SystemC™ [IEE06] based C++ class library providing classes and additional language elements for specifying adaptive digital hardware systems. The modelling approach of OSSS+R is based on the concept of Adaptive Objects (see Section 2.4). Models specified in OSSS+R can be compiled and simulated using a standard C++ compiler and the OSCI SystemC reference implementation. OSSS+R simulations are meant to be cycle-accurate, i.e. they reflect the timing behaviour of a possible hardware implementation.

Like OSSS, also OSSS+R has been created with automatic synthesis in mind and is targeted to partially reconfigurable FPGA platforms such as the Xilinx Virtex 4 [VIR]. In order to reflect this during simulation the library implicitly instantiates and models infrastructure elements which are necessary for an FPGA implementation of a given design. Additionally, the designer can annotate reconfiguration times to reflect the timing behaviour of the physical process of FPGA reconfiguration. In the future, because of its internal modular structure, the library may be extended to other target platforms or to higher abstraction levels by either adding or omitting implementation dependent details and characterisations.

The foundations for the creation of OSSS+R have been laid by the work of the PolyDyn [Pol] project. This ongoing research project investigates the general concept of object-oriented modelling of reconfigurable hardware and provides a proof-of-concept li-

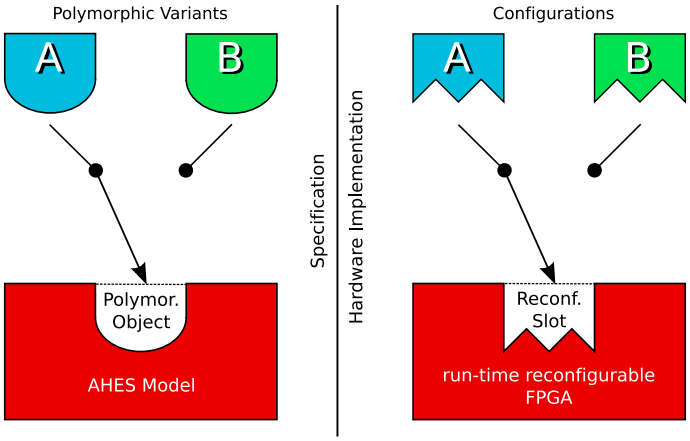



Figure 2.2: Using Adaptive Objects in a model and an FPGA implementation.

brary [SON06]. This library defines the starting point for the development of a stable and industrially applicable modelling library for reconfigurable hardware systems.

## 2.4 Adaptive Objects

Based on the original OSSS approach of combining object-orientation and hardware design, the concept of Adaptive Objects uses object-orientation as an adequate abstraction mechanism for reconfigurable hardware systems. The concept is based on the fact that changing functions of parts of a hardware system largely resembles to the use of polymorphism in object-oriented software design.

Polymorphism, as it is used in object-oriented programming, enables calling methods on an object, whose exact type is unknown to the caller. Only the interface of those methods is defined within the base class of the object. The actual class of the object may be any of the available subclasses derived from the base class. Depending on the actual class of the object, the corresponding implementation of the method is executed. This technique enables changing parts of the software even at runtime without modifying the static part of the code as the interface to the dynamic part does not change.

Considering a digital hardware system consisting of a static and a dynamically reconfigurable part, it is obvious that the interface between the two parts needs to be fixed, i.e. the input/output ports and signals cannot be changed or extended. However, the implemented functionality of the reconfigurable hardware may change. Hence, the key idea of an Adaptive Object is to model the reconfigurable part of a hardware system as polymorphic variants of an object with a fixed interface. This interface is defined by a base class, while the different variants belong to different derived subclasses. During the runtime of a system different implementations of an Adaptive Object can be selected and used (see Figure 2.2).

## 2.5 Convention

During the rest of this document we do not distinguish explicitly between OSSS and OSSS+R anymore. When OSSS+R is mentioned explicitly it is considered to be part of the OSSS simulation library.

# 3 Getting Started

In this chapter we will describe the necessary steps to create a working environment for SystemC™, followed by a short example. So readers already familiar with SystemC can skip the following sections and continue reading at Section 3.3, where we will describe how to get started with OSSS.

## 3.1 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 [gccb] (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 3.1 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 3.1: Compiling SystemC 2.2.0 on a Linux machine using a tcsh C shell

## 3.2 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.

### 3.2.1 Register transfer models

```
1  SC_MODULE(PointlessExample)
2  {
3    sc_in<bool>  pi_bClk, pi_bReset, pi_bMode;
4    // some ports of class type
5    sc_in<Float<8,23> >                  pi_fData;
6    sc_in< Matrix<Float<8,23>, 3, 1> > pi_Vector;
7    sc_out<Matrix<Float<8,23>, 3, 1> > po_Vector;
8
9    // determine the next state
10   void NextState()
11   {
12     ms_bNextState.write(pi_bMode.read() == true &&
13                         pi_fData.read() == 42);
14   }
15
16   // write the next state into the state register
17   void UpdateState()
18   {
19     if (pi_bReset.read() == true)
20       ms_bState.write(false);
21     else
22       ms_bState.write(ms_bNextState.read());
23   }
24
25   // determine output from state and inputs
26   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.

```
27   {
28       if (ms_bState.read() == true)
29         po_Vector.write(pi_Vector.read());
30       else
31         // invoke a method on the object from the input
32         po_Vector.write(pi_Vector.read().myMethod(pi_fData.read()));
33   }
34
35   sc_signal<bool>  ms_bState, ms_bNextState;
36
37   SC_CTOR(PointlessExample)
38   {
39     // combinatorial processes
40     SC_METHOD(NextState)
41     sensitive << pi_bMode << pi_fData;
42     SC_METHOD(Output);
43     sensitive << ms_bState << pi_Vector << pi_fData;
44
45     // sequential process with synchronous reset
46     SC_METHOD(UpdateState);
47     sensitive_pos << pi_bClk;
48   }
49 };
```

Listing 3.2: 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 3.2.

### 3.2.2 Behaviour models

```
1  SC_MODULE(PointlessExample2)
2  {
3    typedef Matrix<sc_fixed<16,1>, 4, 1> Vector_t;
4    typedef Matrix<sc_fixed<16,1>, 4, 4> Matrix_t;
5
6    sc_in<bool>         pi_bClk, pi_bReset;
7    sc_in<bool>         pi_bDataAvail;
8    sc_in<Vector_t>     pi_Vector;
9    sc_out<Vector_t>    po_Vector;
10   sc_out<bool>        po_bDataRdy;
11
12   void Main()
13   { // declare all local variables at the beginning (except
14     // for for-loop indices
15     Matrix_t Transform = Matrix_t::RotZ90;
16     Vector_t result;
17
18     // invoke method to reset all vector elements
19     result.SetElements((Vector_t::BaseType)0);
20
21     wait();
```

```
22      while(true)
23      {
24        do { wait(); } while ( !(pi_bDataAvail == true) );
25        po_Vector.write(pi_Vector.read()*Matrix);
26        wait();
27      }
28    }
29
30    SC_CTOR(PointlessExample2)
31    {
32      SC_CTHREAD(Main, pi_bClk.pos());
33      reset_signal_is(pi_bReset, true);
34    }
35  };
```

Listing 3.3: 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 3.3.

## 3.3 Getting the OSSS Simulation Library

### 3.3.1 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.

### 3.3.2 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 3.1.

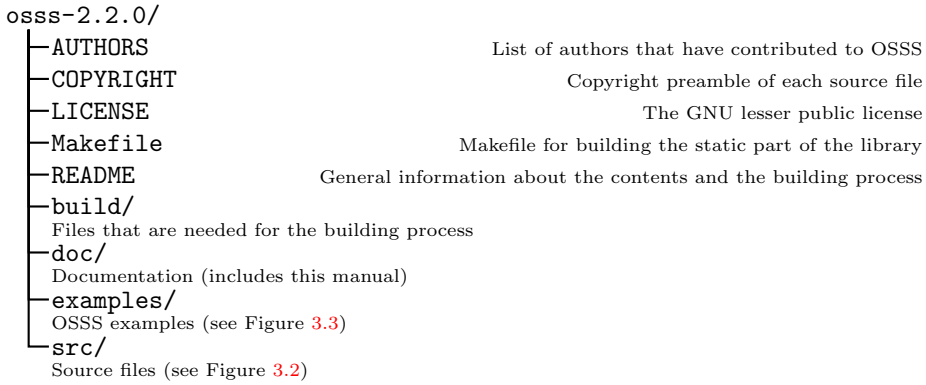



Figure 3.1: Structure of the OSSS 2.2.0 simulation library root directory

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 3.1 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 3.1). Listing 3.4 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 3.4: Messages after successfully configuration of the OSSS library

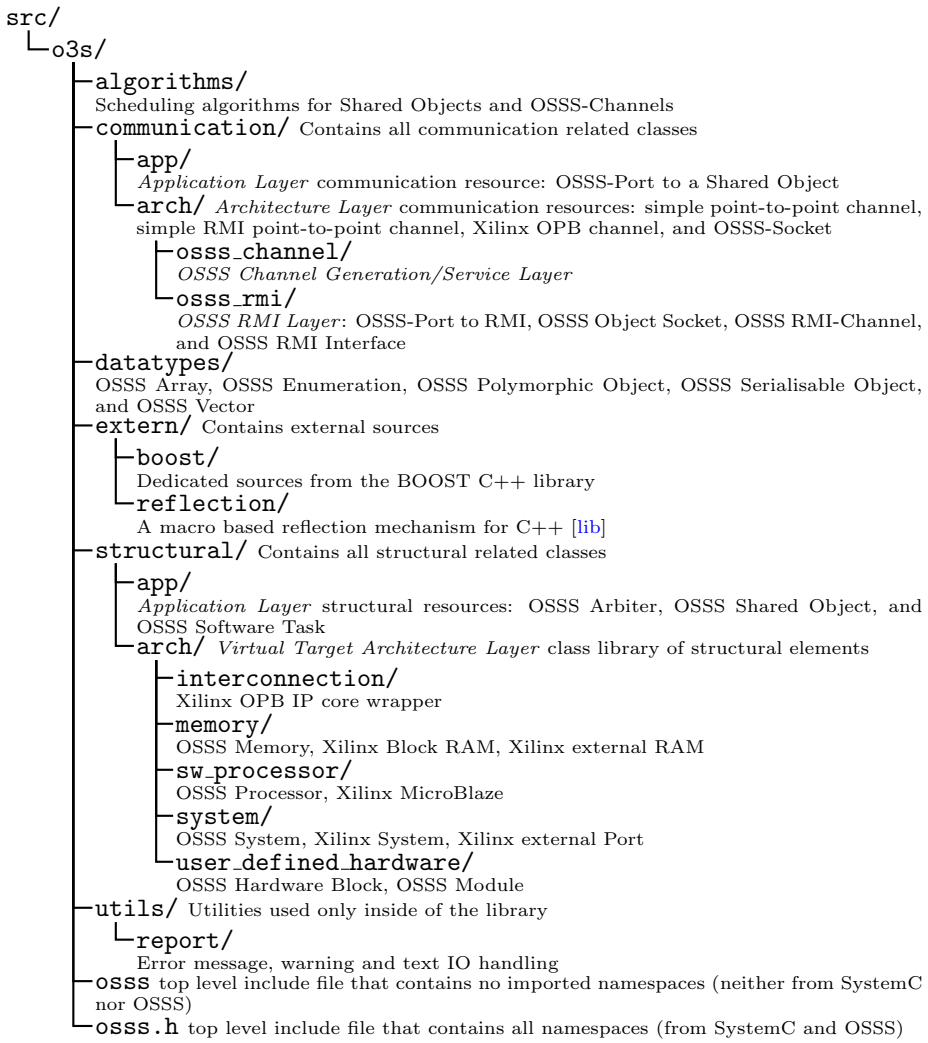



Figure 3.2: Structure of the OSSS 2.2.0 simulation library `src` directory

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 ex-

amples 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 3.3 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 Chapter A.

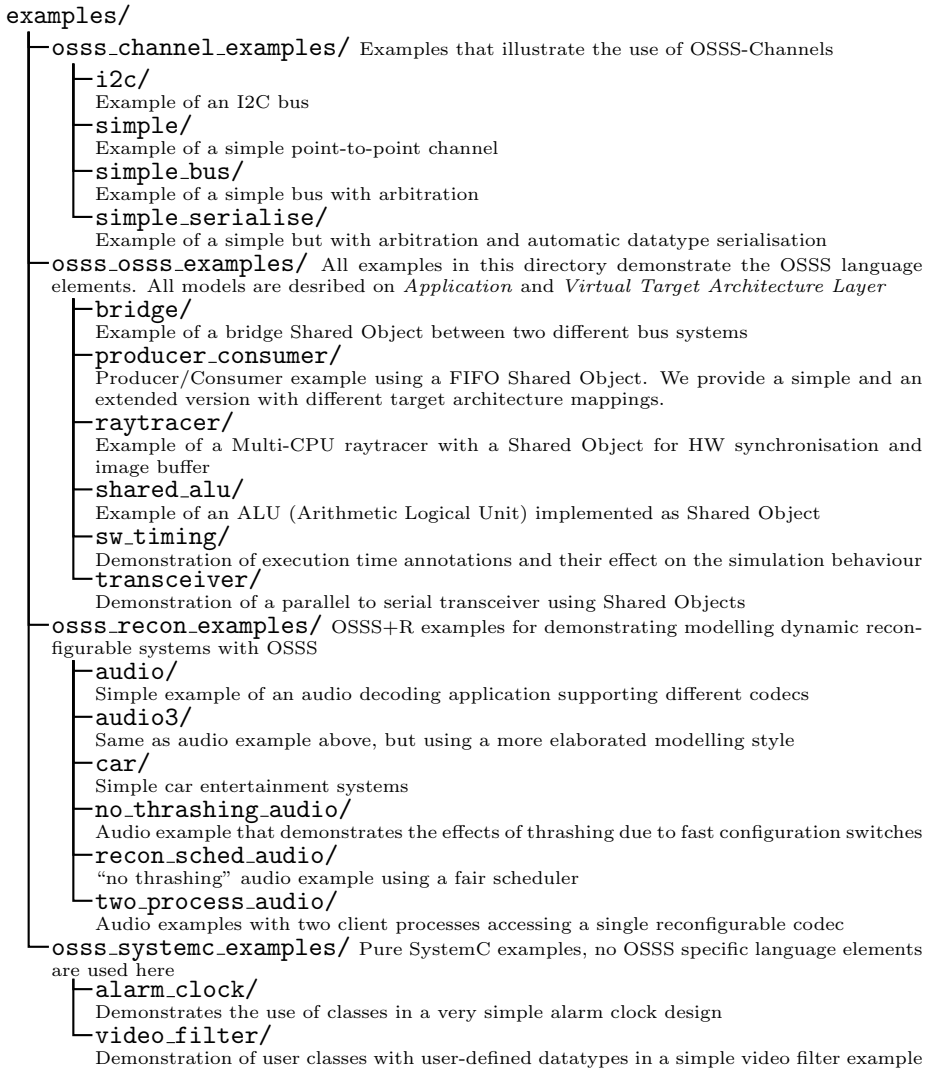



Figure 3.3: Structure of the OSSS 2.2.0 simulation library `examples` directory

### 3.3.3 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 9.1).

# 4 The OSSS Methodology

## 4.1 Introduction

Today, highly integrated embedded systems have a wide range of usage in our daily life, such as telecommunications and automotive. These modern embedded systems consist of several IP blocks plus a few custom components and often use pre-defined technology platforms to implement them. Existing and upcoming technologies offer ever more possibilities to cope with the increasing application requirements. Nevertheless, efficiently designing such systems remains a major challenge since electronic design automation tools and methodologies cannot keep pace.

### 4.1.1 Embedded Systems

In our definition, embedded systems consist out of an arbitrary number of the following components:

- Software components to be executed on software processors which mostly perform control and supervision tasks,
- Hardware components, that may perform computationally intensive or time critical tasks,
- IP (Intellectual Property) components, either 3rd party components or hardware and software components from previous projects,
- Interface components that perform the communication between Hardware and Software.

It has been observed that the integration of these different pre-designed components into an efficiently working system is much more difficult than the design of a single component. This has massively influenced the principle of platform-based design [SV07].

### 4.1.2 Platforms

In practice, most embedded systems are implemented on so-called technology platforms. These platforms provide a basic configuration consisting out of processors, memories, special hardware resources (or accelerators), communication peripherals, and communication resources (like busses, special cross-bar switches or Network-on-Chips) to connect them all together. The application is implemented on top of this platform configuration. Some platforms provide advanced configuration and parameterisation and offer flexibility for the implementation of different applications (e.g. platform FPGAs [xila]).

### 4.1.3 Design Challenges

From a technical point of view the design of embedded HW/SW systems involves all steps from the initial specification of the application - *what is the system supposed to do* - to the efficient implementation on a technology platform - *how is it implemented*. The challenge consists in finding methods for the description and transformation/refinement of the initial specification to an efficient implementation.
From the industrial point of view the requirements for the design of embedded systems are very tight. Targeted development costs and time-to-market needs to be matched, but at the same time functional and non-functional product requirements need to be fulfilled. Consequences for the design process are: Modelling on the highest possible level of abstraction (mastering of complexity), most accurate analysis of system characteristics (prevention of costly re-design cycles), high degree of automation (raising efficiency and prevention of errors), and re-use of pre-existing components. These are just a few consequences, but all of these have been considered during the development of the OSSS methodology.

## 4.2 Related Work

There are several design methodologies available that address the same demands as OSSS. SpecC [GZD+00] is completely refinement driven and introduces lots of useful concepts, like hierarchical behaviours and channels. Since it is based on the C language it does not fit well into the upcoming object-oriented development of SW dominated SoCs. Moreover, it is not well accepted by the industry which favours SystemC™ [IEE06] that has become IEEE standard most recently. instead. Moreover, the upcoming OSCI TLM2 standard [OSC] is becoming very attractive for modelling communication architectures in complex SoCs. There have already been some attempts to use TLM in rapid prototyping [KG05], which are promising but lack of automatic synthesis support. Commercial C/C++ to hardware synthesis tools, like [cyn, cat, agi], allow behaviour level synthesis. However, these tools are well suited for synthesis of complex functional block, but lack of adequate support for system synthesis. More general system level exploration frameworks like Metropolis [DDM+07] are very promising but mostly lack of automatic synthesis support and cannot offer a seamless path to implementation. Work considering Platform-based behavior-level and system-level synthesis [J. 06] aims at the same direction as out work. The main difference to our approach is that we currently do not consider any behavioural synthesis and favour a user driven scheduling for better traceability instead. Moreover, we focus more on the reuse and integration of existing IP componetns through mapping and transactor synthesis.

## 4.3 OSSS Design 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 4.1) 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 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

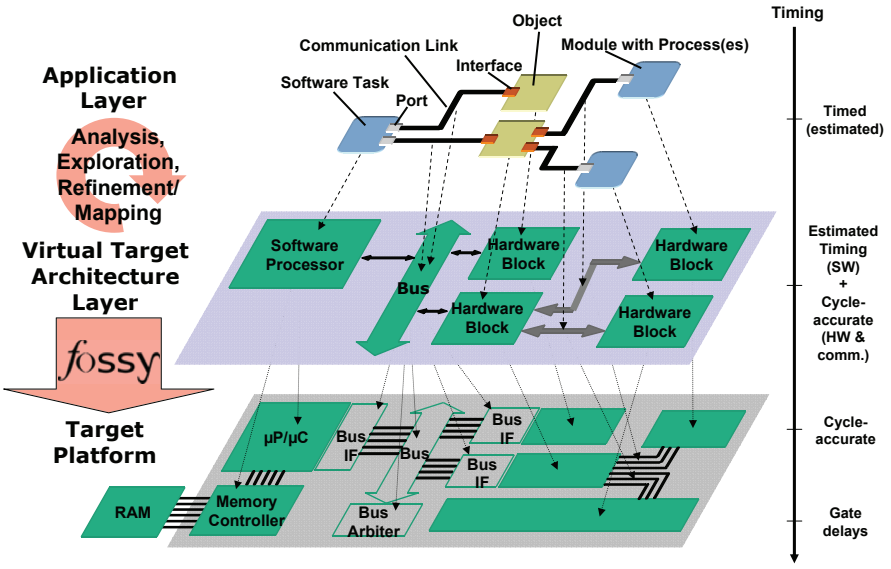



Figure 4.1: OSSS Design Methodology

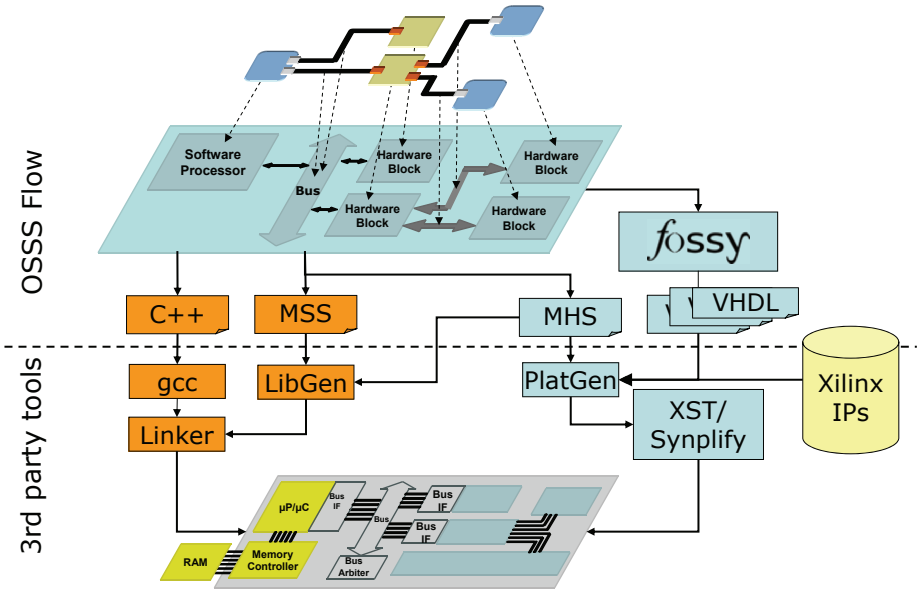



Figure 4.2: *Fossy* synthesis flow for Xilinx FPGAs

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 4.2 shows the *Fossy* synthesis process that has been tailored for Xilinx FPGA target technology.

## 4.4 How does OSSS extend SystemC™?

OSSS extends the synthesisable subset of SystemC and adds constructs that facilitate the use of object oriented features with well-defined simulation- and synthesis-semantics for the description of the hardware part of a hardware/software system. Additionally, OSSS 2.2.0 allows the usage of most of the language constructs provided by C++ when describing the software part of a hardware/software system.

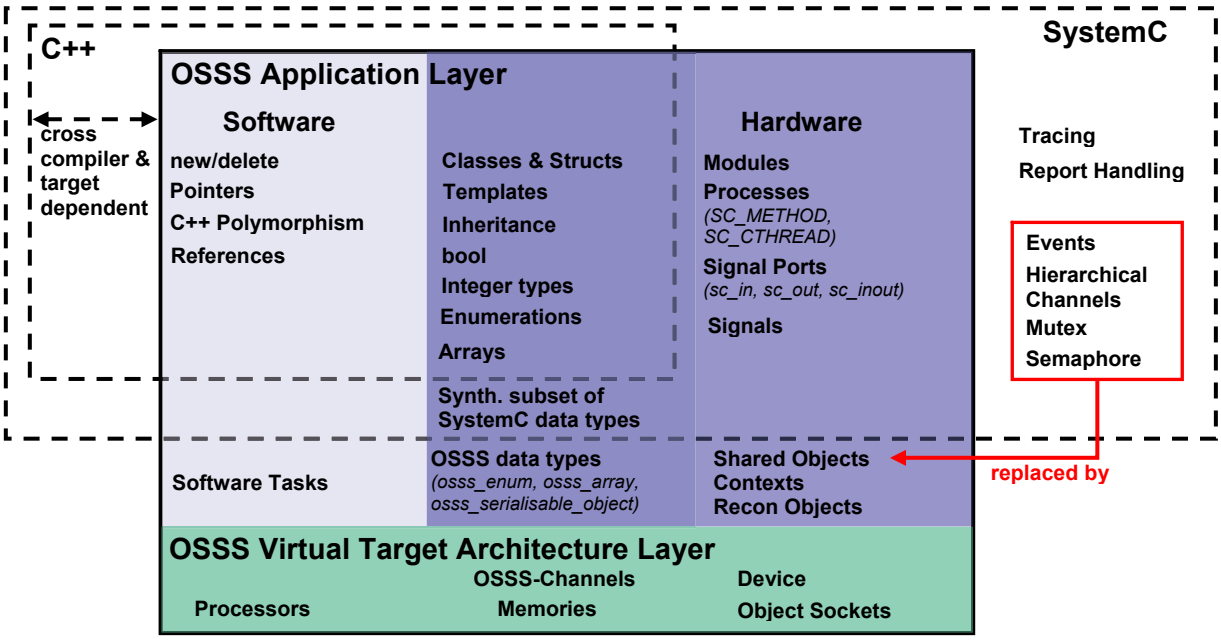



Figure 4.3: OSSS 2.2.0 language subset overview

Figure 4.3 shows the OSSS 2.2.0 language in comparison to C++ and SystemC. Technically, SystemC is a class library that builds on top of the C++ language. From a methodical point of view SystemC extends C++ by a notion of time, concurrent sequential processes, events and signals that are well known from hardware description languages like VHDL or Verilog. The latter point of view is the foundation of the diagram shown in Figure 4.3 where C++ is considered a subset of SystemC. In particular the OSSS 2.2.0 language covers:

- the entire SystemC synthesisable subset (as defined in [Syn04]),
- a part of the C++ language constructs, which are strongly cross compiler and target dependent,
- and extends SystemC by *Software Tasks*, *Shared Objects*, *Polymorphic Objects*, and serialisable data types.
- For the description of a *Virtual Target Architecture* certain architectural building blocks like processors, memories, buses/channels and *Object Socktes* are introduced.

The OSSS 2.2.0 language can be subdivided into a software only, a hardware only, a hardware/software intersection, and a virtual target architecture part. In Figure 4.3 this is illustrated by different colors. In the following paragraphs all important language constructs will be described shortly.

### 4.4.1 The Hardware/Software Intersection

We start with the hardware/software intersection because these language elements can be used in both, the hardware and the software domain. When considering hardware/software systems the data types of this intersection describe the backbone for hardware/software communication.

Structural elements like `structs` and `classes` in both template and non-template versions as known from C++ are allowed. Inheritance (including multiple inheritance) which is a very important feature of object oriented languages is also supported.

Data types are the basic C++ types like boolean (`bool`), integer (`signed char`, `unsigned char`, `char`, `signed/unsigned short`, `signed unsigned int`, ...), and floating point (`float`, `double`). No pointer, reference or "native" array types as known from C++ are supported in this intersection. Please note that the size of the supported basic types are strongly (cross) compiler dependent. Therefore we support the synthesisable subset of SystemC data types[1] as well. These data types allow the specification of well defined sizes. E.g. the `sc_uint<9>` data type has a size of exactly 9 bits after synthesis.

Finally, special OSSS data types are provided:

- `osss_enum<`*`nativeCppEnum`*`>`,
- `osss_array<`*`dataType, Size`*`>` and
- `osss_serialisable_object`.

The `osss_enum<`*`nativeCppEnum`*`>` type serves as a wrapper for native C++ enumeration types (`enum`) and the `osss_array<`*`dataType,Size`*`>` type serves as a wrapper for native C++ array types (e.g. `unsigned int[16]`). The `osss_serialisable_object` is a base class for all user-defined classes that need to be serialised into a bit-vector representation.

### 4.4.2 The Hardware Section

The hardware section consists out of two parts: The "traditional" synthesisable subset of SystemC and special OSSS extensions (*Shared Objects* and *Polymorphic Objects*). The main structural hardware object is a module (`sc_module`) that may contain processes (only `sc_methods` and `sc_cthreads` are allowed *no* `sc_threads`) and signal ports (of type `sc_in<`*`synthDataType`*`>`, `sc_out<`*`synthDataType`*`>` or `sc_inout<`*`synthDataType`*`>`) that can be bound to signals `sc_signal<`*`synthDataType`*`>`.

[1] For more information about the synthesisable data types of SystemC please refer to [Syn04]

Some of the non-synthesisable SystemC features are Events, Hierarchical Channels, Mutexes and Semaphores. These language elements are very useful when designing on a higher level of abstraction, especially at a time where neither the target architecture nor the target platform has been specified. Therefore, OSSS provides a synthesisable replacement for these SystemC features called *Shared Objects*.

*Shared Objects* provide a mechanism to deal with simultaneous method calls and enable modelling of shared resources and abstract communication. They can be customized in different ways and form a synthesisable replacement for Hierarchical Channels, Mutexes and Semaphores. Since they provide a simple guard mechanism that blocks a method call to a *Shared Object* until a certain condition evaluates to true, they can even be used to model events.

There are several other non-synthesisable features of SystemC like signal tracing and report handling. But as these features belong to testbenches that do not need to be synthesisable OSSS does not provide a replacement for them.
Another additional feature of OSSS are *Polymorphic Objects* which allow the use of polymorphism (basically virtual method calls) with a well-defined synthesis semantics. These are considered as a synthesisable replacement for the native C++ polymorphism which is based on pointer semantics. Since pointer types are not synthesisable in hardware the native C++ polymorphism can not be used in the hardware part of an OSSS design.

### 4.4.3 The Software Section

In general the software section of Figure 4.3 contains the whole ISO C++ [cpp98] language. Regrettably, this is only the best-case expectation because we have to mention the capabilities of different software (cross) compilers that reflect the capabilities of the chosen target processor.

The *OSSS Software Task* is the execution environment of the software part of a hardware/-software design. It is the software like counterpart of an `sc_module`. While a `sc_module` can contain multiple processes implemented in hardware an OSSS Software Task contains exactly one single thread of control (i.e. the sequential software program running on it).

### 4.4.4 The Virtual Target Architecture Section

The virtual target architecture is organised as a an extendible *Virtual Target Architecture Class Library* that contains building blocks that need to be instantiated and connected in order to assemble the architecture where the hardware/software application is mapped onto. The architecture class library consists out of software processors, user defined hardware blocks, memories and channels. The OSSS-Channels are used to connect the software processors, the memories and the user defined hardware block with each other.

For simulation purposes all OSSS 2.2.0 language elements are provided by the OSSS simulation library, which is freely available from [sysb].

## 4.5 Conclusion

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

- 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.

# 5 OSSS Modelling Style

## 5.1 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. The general syntax and semantics of the modelling elements are described in the next sections.

## 5.2 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. OSSS+R adopts this approach for the communication between processes and the reconfigurable area.

## 5.3 Synthesis Subset Requirements

For the design of synthesisable hardware systems, OSSS follows the rules of the latest *SystemC Synthesisable Subset* document [Gro04]. It adds capabilities for using objects, e.g. sending objects over signals, and introduces new synthesisable language elements, such as the *Recon Object*. Communication between processes, objects, Shared Objects, and Recon Objects is modelled using method-based interfaces which can be automatically synthesised. The methods are able to call other methods and may contain multiple `wait()` statements. Complete behavioural synthesis is not yet performed, i.e. the scheduling of functional code has to be done manually by the designer.

## 5.4 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. In the future, additional checks may be implemented in an external tool based on the front-end and synthesis tool for OSSS.

## 5.5 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.

# 6 Modelling Elements

## 6.1 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, *Recon Objects* and *Shared Objects* which both can be shared between active parts of the design (see Figure 6.1).

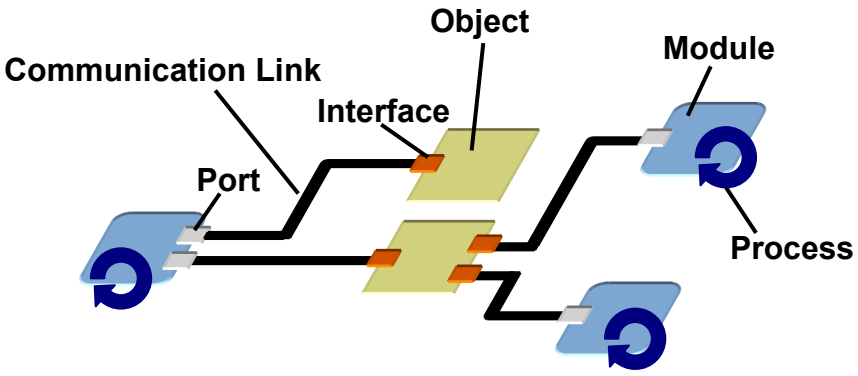



Figure 6.1: The *Application Layer* and its components

Active parts are hardware modules and software tasks that contain processes and thus have an own thread of execution. *Shared Objects* and *Recon Objects* are passive which means they do not initiate any execution on their own. Shared Objects (see Section 6.1.1) 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. Recon Objects (see Section 6.1.6) are special Shared Objects with allow the specification of adaptive behaviour.

Figure 6.2 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 manual.

### 6.1.1 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

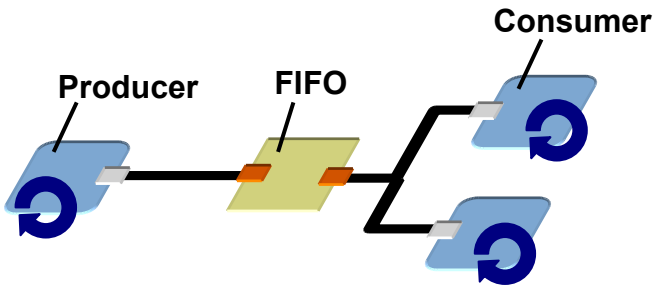



Figure 6.2: Producer/Consumer design on *Application Layer*

synthesisable replacement to model SystemC hierarchical channels, mutexes, semaphores, and events (see red box in Figure 4.3). They are called "shared" because their behaviour is shared between different 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 6.2.

### Instantiation and binding of Shared Objects

Figure 6.3 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 6.1 shows the OSSS Application Layer top-level design of the produer/consumer example as depicted in Figure 6.3. 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");
```

```
24      m_Producer->clock_port(clk);
25      m_Producer->reset_port(reset);
26      m_Producer->output(*m_Buffer);
27
28      m_Consumer[0] = new Consumer("m_Consumer0");
29      m_Consumer[1] = new Consumer("m_Consumer1");
30      for(unsigned int i=0; i<2; ++i) {
31        m_Consumer[i]->clk(clk);
32        m_Consumer[i]->reset(reset);
33        m_Consumer[i]->input(*m_Buffer);
34      }
35    }
36  };
```

Listing 6.1: Top-Level module of the producer/consumer example on Application Layer

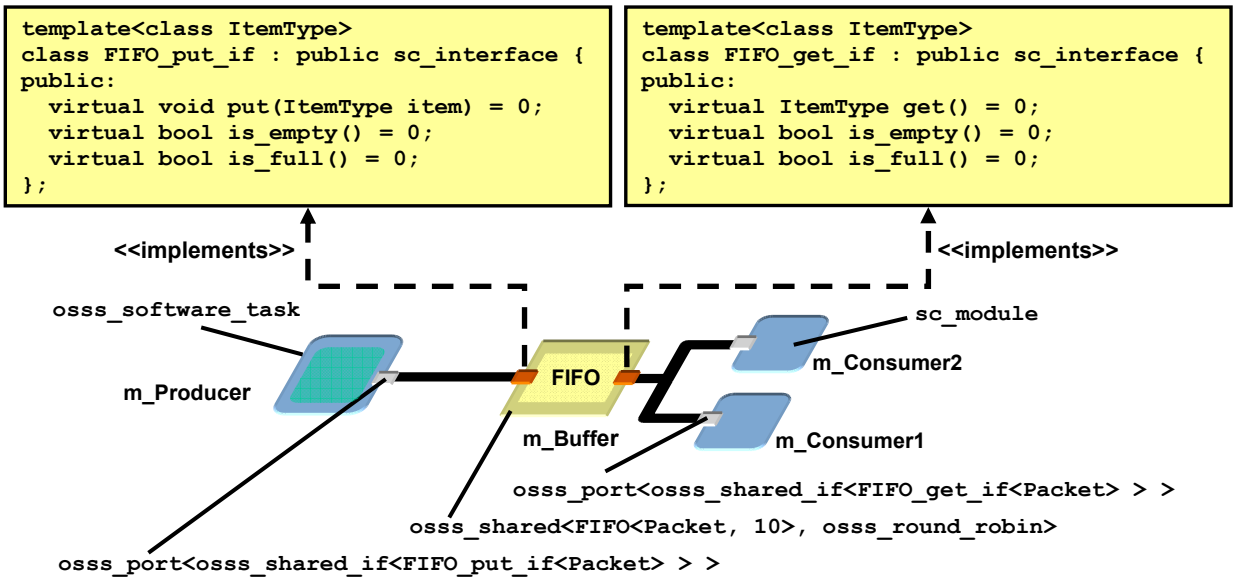



Figure 6.3: Producer/consumer example on the *Application Layer*

### 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 6.2 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
```

```
45   OSSS_GUARDED_METHOD(bool, is_empty, OSSS_PARAMS(0), true) {
46     return m_num_items == 0;
47   }
48
49   OSSS_GUARDED_METHOD(bool, is_full, OSSS_PARAMS(0), true) {
50     return m_num_items == Size;
51   }
52
53  protected:
54
55   void increment_index(FIFO_size_t &index) {
56     if (index == (Size-1)) index = 0;
57     else index += 1;
58   }
59
60   osss_array<ItemType, Size> m_buffer;
61
62   FIFO_size_t m_put_index, m_get_index, m_num_items;
63 };
```

Listing 6.2: FIFO interface and FIFO class implementation

Comparing the FIFO class from Listing 6.2 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 6.1. 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<br>• Fairness not guaranteed |
| `osss_modified_round_robin` | • No priorities<br>• Fairness not guaranteed, "fairer" than unmodified version |
| `osss_static_priority<`*`ZeroIsHighestPrio`*`>` | • Static priorities<br>• Default: Zero is lowest priority<br>• Fairness not guaranteed |
| `osss_ceiling_priority<`*`MaxClients`*`>` | • Dynamic priorities<br>• Fair |
| `osss_least_recently_used<`*`MaxClients`*`>` | • Dynamic priorities<br>• Fair |

Table 6.1: 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.

### 6.1.2 Polymorphic Object

Although SystemC is based on the C++ programming language it does not allow to model hardware using object oriented features like polymorphism. This shortcoming is solved by the OSSS-Library. By using OSSS the designer can write classes with virtual functions and overwrite them in derived classes, like in other object oriented languages. The way objects and method calls are handled differs from the usual C++ style and is necessary to introduce a hardware semantic to Polymorphic Objects.

#### Using Polymorphic Objects

The main difference between the way polymorphism is handled in OSSS and in C++ is that in OSSS polymorphism is not based on pointers, but on "Tagged Objects". Polymorphic Objects provide their own exclusive state space and assignments are done by copying the values of data members rather than changing the reference.

As in C++ calls to virtual function members are dispatched dynamically according to the type of the object they are invoked on.

Next we will illustrate the usage of Polymorphic Objects by taking a look at the "polymorphic_alu" example from the "examples" archive. The idea behind this example is to create a very flexible ALU by using polymorphism. For each operation that the ALU may perform, a new class is derived from the ALUOp class. These derived classes overwrite the `executeOperation()` function to implement the desired behaviour. This way it is possible to create new instructions without altering the ALU itself. The main ALU module just takes the input in form of an ALUOp object, calls the `executeOperation()` function, which gets dispatched dynamically, and writes the result to the output.

Following we will take a closer look at the source code for the polymorphic_alu[1]. Only the header files are regarded here, as the actual implementation is only of lesser interest as it does not differ from the way Polymorphic Objects are implemented in C++.

[1] In all following code excerpts the classes and macros introduced by OSSS are written in bold typeface to allow easier recognition of these new constructs.

```
1  class ALUOp
2  {
3  public:
4    // Each class being used as root class for a Polymorphic Object or being
5    // assigned to a Polymorphic Object must have a OSSS_TAG specifier:
6    OSSS_TAG( ALUOp )
7
8    // Each class being used as root class for a Polymorphic Object or being
9    // assigned to a Polymorphic Object must have a default constructor:
10   ALUOp();
11
12   ALUOp( const sc_bigint< BITWIDTH > & leftOperand,
13          const sc_bigint< BITWIDTH > & rightOperand );
```

Listing 6.3: Excerpt from AluOps.h showing definition of the base class “ALUOp” ( 1/3 )

The code above shows the first lines of code from the class ALUOp. This class is used as a base class for all operations the ALU performs.

Generally polymorphism and inheritance work in OSSS as they do in C++. But there are some formal specifics that are important when using Polymorphic Objects in OSSS. First of all each class that is being used as a base class of a Polymorphic Object or being assigned to a Polymorphic Object must call the OSSS_TAG macro. This way the class is tagged and can be assigned to a Polymorphic Object. Secondly every such class must have a default constructor.

```
1    void setLeftOperand( const sc_bigint< BITWIDTH > & op );
2
3    sc_bigint< BITWIDTH > getLeftOperand() const;
4
5    void setRightOperand( const sc_bigint< BITWIDTH > & op );
6
7    sc_bigint< BITWIDTH > getRightOperand() const;
8
9    virtual void executeOperation( ResultType< BITWIDTH > & result );
10
11   bool operator==( const ALUOp &obj );
```

Listing 6.4: Excerpt from AluOps.h showing definition of the base class “ALUOp” ( 2/3 )

The above excerpt (6.3, 6.4) from the source code shows the prototypes of the ALUOp function members. These function members are either overwritten (e.g. `executeOperation`), or inherited in derived classes. There are function members to set and get the operands and the `executeOperation` function that provides the actual functionality.

```
1  protected:
2    sc_biguint< BITWIDTH > abs( const sc_bigint< BITWIDTH > & val );
3
4    sc_bigint< BITWIDTH > m_leftOperand;
5    sc_bigint< BITWIDTH > m_rightOperand;
6  };
```

Listing 6.5: Excerpt from AluOps.h showing definition of the base class “ALUOp” ( 3/3 )

Finally there is one protected function member, which can only be accessed by classes derived from the ALUOp base class and the data members that provide space to store the operands for every operation.

In the next part we will take a look at one of the derived classes which should help you understand how to derive classes from a base class in OSSS. The following source code shows how the "Add" operation is derived from the "ALUOp" class.

```
1  template< unsigned int BITWIDTH > class Add : public ALUOp< BITWIDTH >
2  {
3  public:
4    OSSS_TAG( Add )
5
6    Add();
7
8    Add( const sc_bigint< BITWIDTH > & leftOperand,
9         const sc_bigint< BITWIDTH > & rightOperand );
10
11   virtual void executeOperation( ResultType< BITWIDTH > & result );
12 };
```

Listing 6.6: Excerpt from AluOps.h showing definition of the derived class "Add"

The class "Add" is derived from the class "ALUOp" the way C++ classes are derived from base classes. Additionally the `OSSS_TAG` macro is called to tag the derived class and allow an assignment to an `osss_polymorphic<>` object. A default constructor is provided along with a constructor that takes the two operands as parameters. The later constructor is implemented to call the corresponding constructor of the base class. The `executeOperation()` function is overwritten to implement the desired behaviour - in this case to add both operands and store the result in the "result" object.

The following listing shows how Polymorphic Objects can be used for module ports. It shows the declaration of the main ALU module.

```
1  template<unsigned int BITWIDTH> SC_MODULE( PolyALU )
2  {
3  public:
4    // port-declarations
5    sc_in_clk    clock;
6    sc_in< bool > reset;
7
8    sc_in< osss_polymorphic< ALUOp< BITWIDTH > > > aluBus;
9    sc_out< ResultType< BITWIDTH > > result;
10
11   void execute();
12
13   //Constructor
14   SC_CTOR( PolyALU )
15   {
16     SC_CTHREAD( execute, clock.pos() );
17     watching( reset.delayed() == true );
18   }
```

```
19 };
```

Listing 6.7: Declaration of the main module of the ALU

To use Polymorphic Objects the variable or port has to be declared of the template type `osss_polymorphic<>`, this template takes the base class as template parameter. Instances of the base class or any class derived of the base class can be assigned to a variable or port declared in this way.

### Restrictions when using Polymorphic Objects

To summarise the above, there are the following restrictions when using polymorphism in OSSS:

- Every base class and every derived class must call the OSSS_TAG macro.
- Every base class and every derived class must have a default constructor.
- A variable that a Polymorphic Object will be assigned to has to be declared of the template type `osss_polymorphic<>`.

The following objects can be assigned to a Polymorphic Object:

- Instances of Polymorphic Object's base class.
- Instances of a class derived from the Polymorphic Object's base class.
- A Polymorphic Object with the same base class.
- A Polymorphic Object with a base class derived from this Polymorphic Object's base class.
- Assigning a Polymorphic Object with a base class which is a parent class for this object's base class requires a cast operation.

Polymorphic Objects in turn can be assigned to:

- Instances of the object's base class.
- Instances of a class from which the Polymorphic Object's base class is derived.
- A Polymorphic Object with the same base type.
- A Polymorphic Object with a base class from which this Polymorphic Object's base class is derived.
- Assigning instance of of classes which are derived from the Polymorphic Object's base class requires a cast operation.
- Apart from these restrictions Polymorphic Objects can be used similar to the way they can be used in C++.

### 6.1.3 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 6.1.1). In contrast to `sc_modules` no nesting of `osss_software_tasks` is allowed.

Before presenting the usage of software tasks by exmaple, Table 6.2 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 6.2: Comparison between `sc_module` and `osss_software_task`

### Declaration of a Software Task

Listing 6.8 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.

```
1  OSSS_SOFTWARE_TASK(my_software_task)
2  {
3
4  public:
5
6    OSSS_SOFTWARE_CTOR(my_software_task)
7    {
8    }
9
10   // alternative constructor
11   my_software_task(...) : osss_software_task()
12   {
13     /* put your software task constructor
14        code here */
15     [...]
16   }
17
18   virtual void main()
19   {
20     /* put your software code here */
21     [...]
22   }
23 };
```

Listing 6.8: Declaration of a Software Task

### Instantiation of a Software Task

Listing 6.9 shows the instantiation of `my_software_task` as declared in Listing 6.8. 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 6.8.

```
1  #define OSSS_BLUE
2  #include <osss.h>
3  #include "my_software_task.h"
4
5  SC_MODULE(Top)
6  {
7  public:
8
```

```
9   sc_in<bool> clk;
10  sc_in<bool> reset;
11
12  my_software_task* mt;
13
14  SC_CTOR(Top)
15  {
16    mt = new my_software_task("my_software_task");
17    // perform binding of special clock and reset ports
18    mt->clock_port(clk);
19    mt->reset_port(reset);
20  }
21 };
```

Listing 6.9: 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 6.10 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 6.10 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 6.11 to see what this message macro writes to the console during the execution of `task1`. Each line in Listing 6.11 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.

```
72 OSSS_SOFTWARE_TASK(task1)
```

```
73 {
74
75 public:
76
77   OSSS_SOFTWARE_CTOR(task1)
78   {
79   }
80
81   void methodX()
82   {
83     PRINT_MSG("Beginning methodX");
84     OSSS_EET(sc_time(3.0, SC_US)) {
85       /* do something else */
86     }
87     PRINT_MSG("Completed methodX");
88   }
89
90   virtual void main()
91   {
92     OSSS_EET(sc_time(2.0, SC_US)) {
93       /* do something */
94       PRINT_MSG("Doing something");
95     }
96     PRINT_MSG("Communication with some other module");
97
98     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
99                   sc_time(2.0, SC_US));
100
101    // Note: The execution time of the initialisation of i and
102    //       for checking the condition (at least the first
103    //       time) is neglected here
104    for (int i=0; i<3; ++i)
105    {
106      OSSS_EET(sc_time(5.0, SC_US)) {
107        PRINT_MSG("For loop, iteration " << i);
108
109        if (i%2 == 0)
110        {
111          // will be called for i==0 and i==2
112          methodX();
113        }
114      }
115
116      EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
117                    sc_time(2.0, SC_US)+
118                    sc_time(5.0, SC_US)*static_cast<double>(i+1)+
119                    sc_time(3.0, SC_US)*((i==2) ? 2.0 : 1.0)
120                   );
121    }
122  }
123 };
```

Listing 6.10: Usage of EETs in a Software Task

Listing 6.11 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 6.11: Console output after running the Software Task from Listing 6.10 (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 6.12 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   }
```

```
142
143   virtual void main()
144   {
145     PRINT_MSG("Beginning time critical calculation");
146     OSSS_RET(sc_time(10.0, SC_US)) {
147
148       OSSS_RET(sc_time(4.0, SC_US)) {
149
150         PRINT_MSG("Beginning time critical sub-calculation 1");
151         OSSS_EET(sc_time(2.0, SC_US)){
152           /* do something */
153         }
154         PRINT_MSG("Completed time critical sub-calculation 1");
155
156         EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
157                       sc_time(2.0, SC_US));
158
159       }
160
161       EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
162                     sc_time(2.0, SC_US));
163
164       PRINT_MSG("Beginning time critical sub-calculation 2");
165       OSSS_EET(sc_time(2.0, SC_US)) {
166         /* do something */
167       }
168       PRINT_MSG("Completed time critical sub-calculation 2");
169
170       EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
171                     sc_time(4.0, SC_US));
172
173     }
174
175     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
176                   sc_time(4.0, SC_US));
177
178     PRINT_MSG("Completed time critical calculation");
179
180     PRINT_MSG("Beginning time critical calculation 2 (which will fail)");
181     OSSS_RET(sc_time(3.0, SC_US)) {
182
183       for (int i=0; i<3; ++i)
184       {
185         OSSS_EET(sc_time(2.0, SC_US)) {
186           PRINT_MSG("For loop, iteration " << i);
187         }
188         EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
189                       sc_time(4.0, SC_US)+
190                       sc_time(2.0, SC_US)*static_cast<double>(i+1));
191
192       }
193     }
194
195     // The previous RET is intentionally violated by inner EETs.
196     // Hence we expect now == 10.0 us instead of 7.0 us
197     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
```

```
198                     sc_time(10.0, SC_US));
199
200     PRINT_MSG("Completed time critical calculation 2");
201
202     PRINT_MSG("Beginning time critical calculation 3 (which is
            inconsistently constrained)");
203     OSSS_RET(sc_time(5.0, SC_US)) {
204       methodY();
205     }
206
207     // The previous RET is intentionally violated by an inner RET.
208     // Hence we expect now == 10.0 us instead of 7.0 us
209     EXPECTED_TIME(sc_time(110.0, SC_NS)+ //reset time
210                     sc_time(14.0, SC_US));
211
212     PRINT_MSG("Completed time critical calculation 3");
213   }
214 };
```

Listing 6.12: Usage of EETs and RETs in a Software Task

Listing 6.13 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 6.13: Console output after running the Software Task from Listing 6.12 (Clock period is 10.0 ns, number of reset cycles is 10)

### 6.1.4 Hardware/Software Communication

Listing 6.14 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 6.1. 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 6.14 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 6.1.1. 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.

```
1  class Producer : public osss_software_task
2  {
3  public:
4
5    // connection to the shared object
6    osss_port<osss_shared_if< FIFO_if<Packet> > > output;
7
8    // runs only once in the beginning
9    OSSS_SW_CTOR(Producer) { }
10
11   // has to override the virtual main()
12   void main()
13   {
14     Packet p;
15     while(true)
16     {
17       OSSS_EET(sc_time(50.0, SC_NS)) {
18         /* some calculations that take approximately
19            50.0 nano seconds */
20       }
21
22       // communication with the "outside world"
23       output->put(p);
24
25       OSSS_EET(sc_time(10.0, SC_NS)) {
26         /* some calculations that take approximately
27            10.0 nano seconds */
28       }
29     }
30   }
31 };
```

Listing 6.14: 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 6.4 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 6.4.

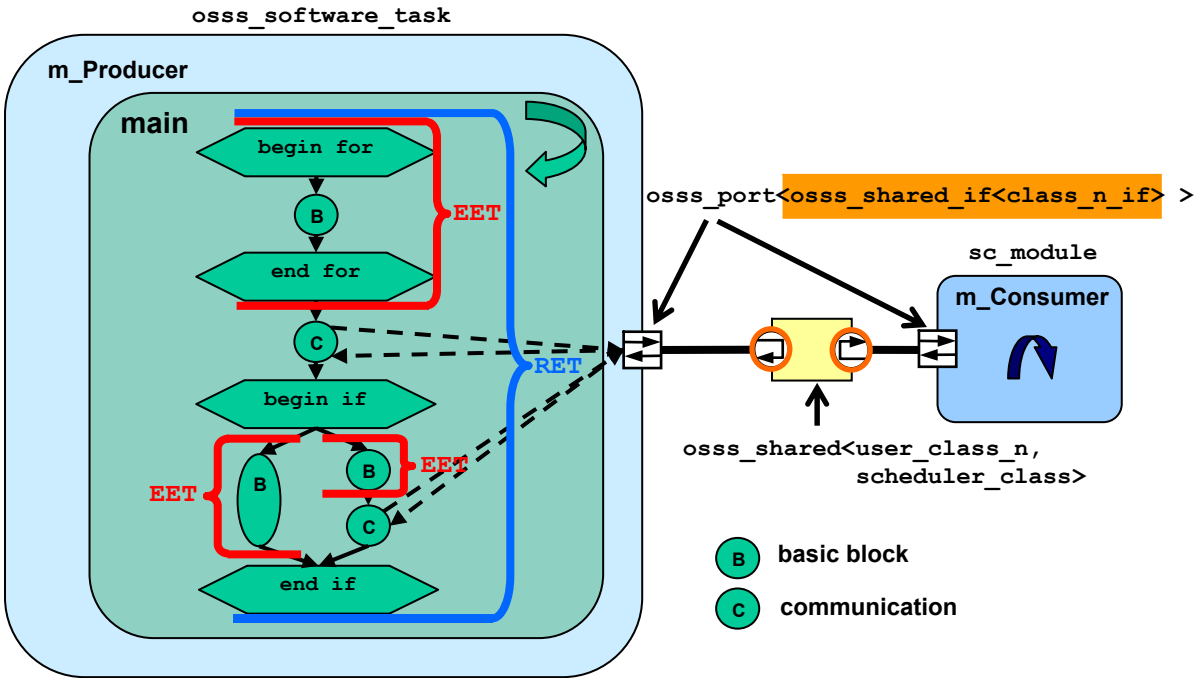



Figure 6.4: EET statements inside of the `main()` method of a software task

Listing 6.15 has the same block structure using EET and RET annotation as shown in Figure 6.4. 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:
```

```
7   virtual void main() {
8     const unsigned char source_addr = 42;
9     unsigned char target_addr = 0;
10    unsigned int offset = 0;
11    Packet p;
12
13    while(true) {
14      OSSS_RET(sc_time(2000.0, SC_NS)) {
15        OSSS_EET(sc_time(120.0, SC_NS)) {
16          p.set_source_addr(source_addr);
17          p.set_target_addr(target_addr);
18          for(unsigned int i = 0; i<p.get_payload_size(); ++i)
19            p.set_payload(i, i+offset);
20        }
21        output->put(p);
22        if (target_addr >= 10) {
23          OSSS_EET(sc_time(10.0, SC_NS)) {
24            target_addr = 0;
25            offset = 0;
26          }
27        }
28        else {
29          OSSS_EET(sc_time(30.0, SC_NS)) {
30            target_addr += 1;
31            offset += 10;
32            p.set_target_addr(target_addr);
33          }
34          output->put(p);
35        }
36      }
37    }
38  }
39 };
```

Listing 6.15: Producer Software Task

Listing 6.16 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.

```
1  class Packet {
2   public:
3
4    Packet();
5
6    unsigned char get_source_addr() const;
7    void          set_source_addr(unsigned char addr);
8
9    unsigned char get_target_addr() const;
10   void          set_target_addr(unsigned char addr);
11
12   unsigned char get_payload(unsigned int index) const;
13   void          set_payload(unsigned int index,
14                             unsigned char data);
```

```
15
16   unsigned int   get_payload_size() const;
17
18  protected:
19   unsigned char   m_source_addr;
20   unsigned char   m_target_addr;
21   unsigned char   m_payload[10];
22 };
```

Listing 6.16: Signature of the `Packet` class

### 6.1.5 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 6.17: Consumer HW module implementation

### 6.1.6 Recon Object

The *Recon Object* is the central modelling element in OSSS+R. It implements the concept of an Adaptive Object by representing a reconfigurable area which can be alternately used by different functional blocks. It provides a connection point between the static and the adaptive part of a design.
The Recon Object is a structural container object which can hold other objects and automatically instantiates and provides necessary infrastructure elements. *Structural* means that it is different from an ordinary object (e.g. a collection of functions or data which can be created, copied, and arbitrarily passed around), but similar to a SystemC module which is unique and has a fixed position within the design hierarchy.
The method interface of the Recon Object is statically determined by an interface class which is provided as a parameter to the Recon Object at its instantiation. Calls to those interface methods are automatically delegated to the currently contained object. An ordinary object is put into a Recon Object by assignment. This implies that every object that is assigned to a Recon Object needs to implement the given interface.
To realise polymorphism, the types of the assigned objects are allowed to be derived from the original base class. Thus, they may overload the base class methods with different implementations. Depending on the type of a currently assigned object, different functional implementations may be executed when calling a method on the Recon Object. Therefore, objects of different type represent different types of hardware blocks. In contrast, objects of the same type use the same type of hardware, but may have different states, i.e., their attributes have different values.
From the designer's point of view, the Recon Object behaves similar to a base class pointer in C++. A base class pointer may point to different objects that have different subtypes (derived from the base class). A method call to the base class pointer automatically calls the method implementation that is associated with the subtype of the current object.
A Recon Object is a shared object which can be accessed concurrently by more than one process at a time. To resolve possible conflicts the Recon Object automatically instantiates an access controller which serialises incoming requests. The serialisation is done using a built-in scheduler. This scheduler may either be a provided standard scheduler or a user-defined scheduler. Besides the serialisation of concurrent accesses, the access controller also detects the need for a reconfiguration and performs the necessary steps to process the reconfiguration. Although the behaviour of the access controller is completely modelled and reflected during simulation its internals are completely transparent to the designer.

#### Instantiating a Recon Object

The functionality of a Recon Object is realised in the OSSS+R library using the container class `osss_recon`. A Recon Object is created by instantiating an object of this class as a submodule of an `SC_MODULE`. In Listing 6.18 an object named `my_recon` of type `osss_recon<my_if>` is instantiated using `my_if` as the interface parameter.

```
1 class my_if : public osss_object {
2  public:
3   virtual int my_function(const int& i) { return 0; }
4   my_if() {
5     OSSS_BASE_CLASS(osss_object); //mandatory OSSS+R macro
6   }
```

```
};

SC_MODULE(my_module) {
  ...
  osss_recon<my_if> my_recon;
  ...
};
```

Listing 6.18: Instantiating a Recon Object

The interface class `my_if` is defined before the instantiation of `my_recon` and includes one method named `my_function`. The method has an integer as input parameter and returns an integer after completion. Note that there are two required OSSS+R language elements which are needed for implementation reasons. The interface class needs to be derived from `osss_object` and the designer has to add the `OSSS_BASE_CLASS` macro to the constructor. This macro indicates the direct base class of the defined class. It is mandatory for all classes that are used in conjunction with a Recon Object.
All interface methods should be declared to be virtual using the keyword `virtual`. In contrast to C++ the interface methods of the base class in OSSS+R are not allowed to be pure virtual. They need to provide a default implementation that is automatically called if no object is currently assigned to the Recon Object.

### Assigning Objects

Objects are assigned to a Recon Object using the `=` operator. All assignments have copy semantics. Hence, the content of an assigned object is copied to the Recon Object while the original object remains unchanged. In the case of an assignment using the constructor of a class the object is directly instantiated within the Recon Object.
Listing 6.19 shows the definition of the derived class `my_class`, the instantiation of an object `my_object`, and its assignment to `my_recon`. The last line shows an assignment using the default-constructor of `my_class`.

```
class my_class : public my_if {
 public:
  virtual int my_function(const int& i) { return i+j; }
  my_class(const int& s = 1) : j(s) {
    OSSS_BASE_CLASS(my_if); // mandatory OSSS+R macro
  }
 private:
  int j;
};

my_class my_object; // object instantiation

my_recon = my_object; // object assignment
my_recon = my_class(); // direct object instantiation
```

Listing 6.19: Assigning objects to a Recon Object

Every new assignment to a Recon Object leads to the destruction of the previously assigned object. In general, one could preserve the state of the previous object by manually copying the content of the Recon Object to a temporary object and restoring the state later on by

assigning the temporary object back to the Recon Object. However, to achieve object persistence throughout consecutive reconfigurations it is recommended to use *Named Contexts* (see Section 6.1.7).

### Calling Methods

Interface methods can be called directly on a Recon-Object via the arrow operator as known from C++. The simulation library automatically passes this call to the internal access controller. Calling a method of an object within a Recon Object is shown in Listing 6.20. The previously required `osss_call()` wrapper is no longer needed in the current version of OSSS+R.

```
1  my_recon = my_class();
2
3  // deprecated use of osss call
4  // int result = osss_call(my_recon)->my_function(5);
5
6  // correct use of operator->(...)
7  int result = my_recon->my_function(5);
```

Listing 6.20: Calling a method

In this version of OSSS+R all method calls are blocking, i.e. the calling process is blocked until the method has been executed and terminated. Besides of the execution time additional blocking time may arise due to the scheduling of the access controller. There are, however, plans to extend the methodology to non-blocking calls in the future.

### Process and Port Binding

The clock and reset ports of the Recon Object need to be bound before the start of a simulation. The port binding is done similar to ordinary SystemC ports and signals. Usually, the designer will bind the ports to the clock and reset ports of the parent SystemC module. Although other bindings are possible, the current implementation requires that all modules and processes that are accessing a Recon Object need to be driven by the same clock and reset.
For synthesis reasons, all processes that are accessing a Recon Object need to be registered to the Recon Object. This is done, using a `osss_uses`-call directly after the process declaration in the module constructor (similar to a `sensitive` or `reset_signal_is` statement). A complete binding of `my_recon` is shown in Listing 6.21.

```
1  SC_MODULE(my_module)
2    sc_in<bool> clk;
3    sc_in<bool> res;
4    ...
5    SC_CTOR(my_module) {
6      my_recon.clock_port(clk);
7      my_recon.reset_port(res);
8
9      SC_CTHREAD(my_process, clk.pos());
10     reset_signal_is(res);
11     osss_uses(my_recon);
12   }
13   ...
```

```
14 };
```

Listing 6.21: Port and Process Binding

### 6.1.7 Named Context

The Recon Object provides a way to model and use reconfiguration within a hardware design. However, having only the Recon Object as an abstraction mechanism still has some issues:

- The reconfiguration is modelled explicitly, because the designer triggers reconfiguration directly by assigning objects to the Recon Object.
- Object persistence throughout consecutive reconfiguration (if required) has to be handled manually by the designer by copying and restoring the content of a Recon Object.
- Concurrent access to a Recon Object may yield race conditions if multiple processes are trying to using different objects at the same time.

As an approach to use the reconfiguration transparently and in order to ensure object persistence automatically, we introduce the concept of *Named Contexts*. This concept loosely resembles process contexts on multitasking systems where multiple processes are sharing one processor. Here, a context refers to all information that is necessary to execute, suspend, and resume a process, such as instruction and data segments, program counter, stack pointer, and others. A context switch is the transfer of the processor from one process to another. This includes saving the context of the previous process and restoring the context of the next process.
Similarly, a *Named Context* in OSSS+R includes all information related to an object associated with a Recon Object. In contrast to an object that is directly assigned to a Recon Object, the object that is indicated by a Named Context does not have to reside within the Recon Object all the time. Instead it is only configured when it is accessed through a method call. Accessing an object by the use of a Named Context ensures persistence through consecutive reconfigurations of the Recon Object and resolves conflicts resulting from concurrent accesses to different objects.
The handling of all Named Contexts associated with one Recon Object is done by the Recon Object itself and is completely transparent to the designer. That is, the Recon Object automatically detects the need for a context switch, decides and initiates the type of reconfiguration, and takes cares of saving and restoring context states.
A context switch takes place if a Named Context other than the currently enabled one is accessed by a process. There can be two types of a context switch:

- A switch between objects of the same type – only the objects' attributes need to be exchanged.
- A switch between objects of different types – the hardware has to be reconfigured.

In both cases, the attributes of the enabled context are saved to the attribute storage and the attributes of the next context are restored. A reconfiguration of the hardware is done (i.e. simulated by OSSS+R) only if the class types of the involved objects are different.
Named Contexts need to provide at least the interface of the Recon Object they are associated with. Thus, all methods that can be directly called on the Recon Object, can also be called on the Named Context. This implies that the interface class of a Named Context needs to be either the interface class of the Recon Object or a subtype of that interface. Named Contexts are also allowed to provide specialised interfaces, i.e. their interface class may provide more methods than the base class interface of the Recon Object. In that case, only those objects can be assigned to the Named Context whose class is either that specialised class or is derived from that class.
Besides the Named Contexts the OSSS+R methodology knows of two other types of contexts which are generalisations of the Recon Object approach. The *Anonymous Context* is the actual context that is accessed if a process calls a method on the Recon Object instead of a Named Context. The *Temporary Context* is created if a process directly assigns an object or a constructor to the Recon Object. It is only enabled as long as no other object is assigned and no Named Context is accessed. In these cases the Temporary Context is destroyed and cannot be restored. Note, that the mixed usage of Temporary Contexts and Named Contexts may result in undetermined behaviour, if more than one process is accessing the Recon Object.

### Creating Named Contexts

Listing 6.22 shows how to define a Named Context using the `osss_context` class and the interface parameter `my_if`.

```
1 osss_context<my_if> my_context;
```

Listing 6.22: Creation of a Named Context

As previously stated the interface parameter is not limited to the interface of the Recon Object. However, the interface always needs to be derived from this base class interface. The extreme case would be a Recon Object that has `osss_object` as base class interface (such that it does not provide any methods itself) and Named Contexts whose interfaces are derived from `osss_object`. This allows for using a Recon Object with unrelated classes and objects.

### Assigning Objects

Objects are assigned to a Named Context similar to the assignment to a Recon Object. Assigning an existing object to a Named Context results in a copy of that object which will be created within the associated Recon Object. If a constructor is assigned, the object will directly be instantiated within the Recon Object. Listing 6.23 demonstrates the corresponding types of assignments.

```
1 my_class my_object; // object instantiation
2 // ...
3 my_context = my_object(); // object assignment
4 my_context = my_class(); // direct instantiation
```

Listing 6.23: Assigning objects to a Named Context

Similar to Recon Objects, consecutive assignments to a Named Context result in the destruction of the previous object. However, different to Recon Objects, the designer may create multiple Named Contexts used with one Recon Object, to have more than one object at a time.

### Calling Methods

Calling a method on a Named Context is similar to a method call on a Recon-Object. Instead of the Recon-Object the arrow-operator of the Named Context can be used directly (see Listing 6.24).

```
1 my_context = my_class();
2 int result = my_context->my_function(5); // method call
```

Listing 6.24: Calling a method on a Named Context

### Context Binding

Named Contexts need to be statically bound to one Recon Object. To preserve well-defined and efficient synthesis semantics they are not allowed to migrate from one Recon Object to another. Also, there has to be a binding between processes and Named Contexts. Listing 6.25 shows the bindings for a Named Context within the constructor of a module.

```
1  SC_MODULE(my_module)
2    sc_in<bool> clk;
3    sc_in<bool> res;
4    //...
5    SC_CTOR(my_module) {
6      //...
7      my_context.bind(my_recon); // Binding to Recon Object
8
9      SC_CTHREAD(my_process, clk.pos());
10     reset_signal_is(res);
11     osss_uses(my_context); // binding to process through context
12     // osss_uses(my_recon);   // instead of using the bound Recon Object
13   }
14   //...
15 };
```

Listing 6.25: Binding of a Named Context

Because a Named Context is always bound to a Recon Object, a binding of a process to a Named Context implies a binding of the process to the Recon Object. Thus, the explicit binding to the Recon Object can be omitted, but binding the clock and reset ports of the Recon Object is still required.

## 6.1.8 Advanced Reconfiguration Control

A typical issue for partially dynamically reconfigurable hardware systems is the high amount of time needed to perform a reconfiguration. For instance, partial reconfiguration of a Xilinx Virtex4 FPGA might take between 10 or 100 ms, depending on the size of the reconfigured area. This usually leads to several problems. For multi-tasking applications,

where a set of mutually exclusive modules are mapped to the same FPGA area, continuous switching between these configurations might result in situations where most of the operation time of the system is spent for reconfiguration, leading to a bad ratio of computation to configuration. This so-called *configuration trashing* might be reduced by adopting the chosen scheduling strategies that best match the specific application.
Within the OSSS+R methodology, there are several ways to cope with such a situation, some of which will be introduced in the following sections. Another approach to reduce this problem using adaptive locks based on synchronisation through Shared Objects has been published recently in [GON08].
On the other hand, in some types of applications, like signal processing, interruption of the computation process might not be possible at all, i.e. if data cannot be buffered or no output latency is allowed. To deploy dynamic reconfiguration even in such scenarios, the prefetching pattern is introduced in Section 7.2.

### Changing the scheduling policy

In OSSS+R, configuration trashing occurs, when accesses from different processes to Contexts bound to the same Recon Object result in reconfigurations very often. This is due to the fact, that after each method-call on a Named Context, the Access Controller of the Recon Object has to schedule the next job out of the current pending requests. Since the default scheduler is a fair, round-robin based scheduler, it is not uncommon that a reconfiguration is required before each request.
An obvious approach to tackle this problem is the change of the used scheduling policy. This can be done during either by selecting one of the existing scheduler classes in the modelling library, or via defining an (maybe application-specific) scheduling class, derived from `osss_recon_scheduler`. Listing 6.26 shows how to specify a custom scheduler during the elaboration of a system.

```
1  class my_scheduler
2    : public osss_recon_scheduler
3  {
4    /* ... */
5    osss_job_idx schedule(); // choose the next job
6  };
7
8  SC_MODULE(my_module)
9  {
10   osss_recon< my_if > recon;
11   //...
12   SC_CTOR(my_module) {
13     //...
14     recon.setScheduler< my_scheduler >();
15   }
16   //...
17 };
```

Listing 6.26: Choosing a custom scheduling policy

Although the scheduler class can include its own custom members, that are preserved between different scheduling requests, it might have sufficient knowledge about the overall situation of the calling processes. Therefore, the processes themselves can influence the scheduling by using external locks as described in the next section.

### Context Locking

Another way to suppress competing reconfiguration requests by different processes on a single Recon Object is the use of external locks. OSSS+R provides two block annotations to enforce certain locking strategies.

**OSSS_KEEP_ENABLED()** Wrapping a part of a process inside an `OSSS_KEEP_ENABLED()` block ensures, that the Named Context that is given as an argument is not disabled (and therefore replaced through either class or context switch see Section 6.1.7) as long as the process' thread of control is within the current code block.

```
1 OSSS_KEEP_ENABLED( ctx ) // request an enabled context
2 {                        // (potentially blocking)
3   while( input_data_available ) {
4     // consecutive accesses WITHOUT reconfiguration
5     ctx->process( input_data );
6   }
7 } // context may become disabled again
```

Listing 6.27: Keeping a context enabled

Since this block annotation only ensures, that the given context stays *enabled*, other processes may still use the same context. The Access Controller of the underlying Recon Object can schedule concurrent requests for *this* context, since the permission is always returned after each request. Only requests to other contexts are blocked and therefore the number of reconfigurations due to configuration trashing can be reduced, as exemplified in Listing 6.27. On entering an `OSSS_KEEP_ENABLED()` block, the current process tries to enable the given context *immediately*. Therefore, the entrance is potentially blocking, e.g. if the enabling the requested context requires a reconfiguration. When the block is left, the Access Controller is informed about the release of the external lock.

**OSSS_KEEP_PERMISSION()** Usually, the granted permission for a Named Context or Recon Object request is returned immediately after the corresponding action is finished. This ensures, that multiple clients can access the Context (and maybe even the shared Recon Object) as soon as possible. If a process uses a context exclusively, it may even decide to keep the granted permission throughout multiple requests, denoted by an `OSSS_KEEP_PERMISSION()` annotation.
Keeping the permission obviously keeps the Named Context enabled as well (cf. `OSSS_KEEP_ENABLED()`), since the Access Controller is not allowed to disable a context while a process is accessing it. Furthermore, subsequent requests do not have to request the permission again, which reduces the communication overhead implied by the permission handling. This might further increase application throughput and can be used to model atomic sequences of several accesses to a given context.
In case of multiple accessing processes, it has to be taken care of potential deadlocks due to the client-side locking. Therefore, the use of this annotation is recommended mainly for exclusive usage of the corresponding contexts.

```
1 osss_recon<>                recon;
2 osss_context< user_class > ctx1, ctx2; // bound to recon
3 // ...
4 OSSS_KEEP_PERMISSION( recon ) // change permission handling of recon
```

```
5   {                                   //   (not blocking)
6     // do something with the contexts
7     ctx1 = some_class();              // blocking, until permission received
8     ctx1->some_method();              //   we still have the permission
9
10    // do something with another context
11                                      // return kept permission of ctx1
12    ctx2->do_something();             // request permission of ctx2
13                                      //    (scheduler kicks in here!)
14    ctx2->do_more();                  // here, we still have the permission
15
16  }                                   // unset "keep permission" mode
```

Listing 6.28: Choosing a custom scheduling policy

Since this "lazy" permission handling is implemented completely within the client process, entering an `OSSS_KEEP_PERMISSION()` has no visible impact on the timing. The permission handling can be set for a given context only, or for a Recon Object as a whole. In the latter case, which is depicted in Listing 6.28, the permission always held for the context accessed last and only returned during an access to another context or if the annotated block is left.

## 6.2 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.

### 6.2.1 Architecture Class Library

Figure 6.5 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 6.5 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[2].
All architecture building blocks in Figure 6.5 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 clock and reset signal. There is no semantical difference between the `sc_module` and the `osss_module`.

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

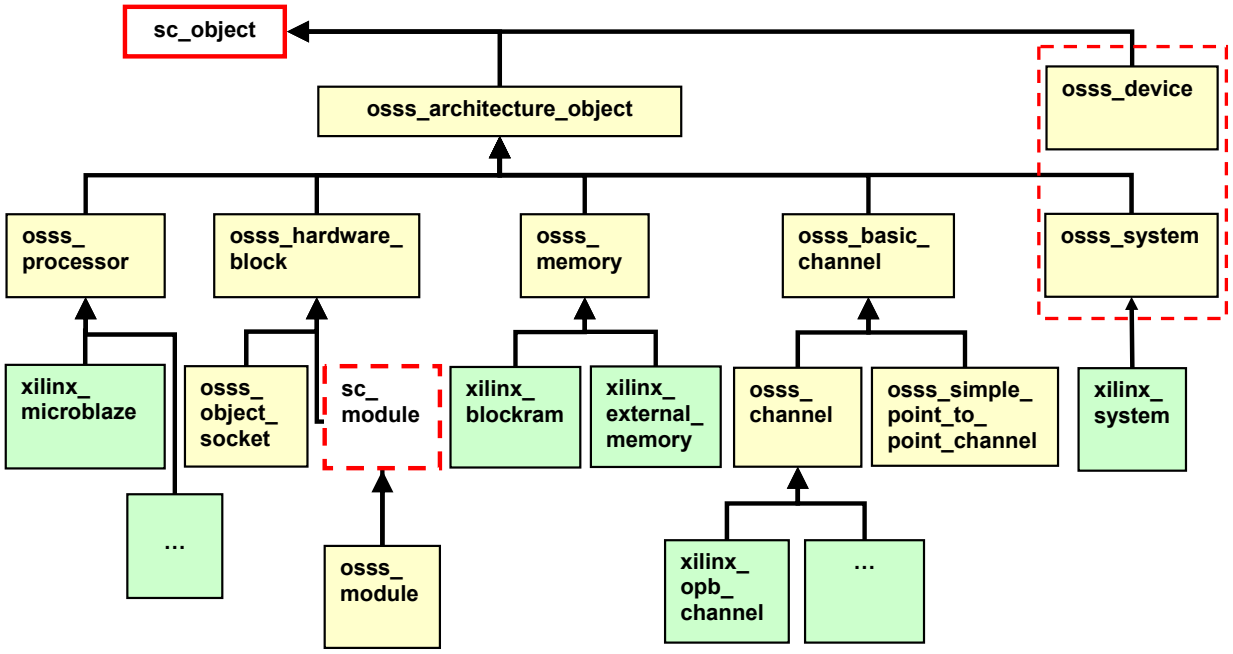



Figure 6.5: Sample of the OSSS Architecture Class Library

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 Section 6.2.5.
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.

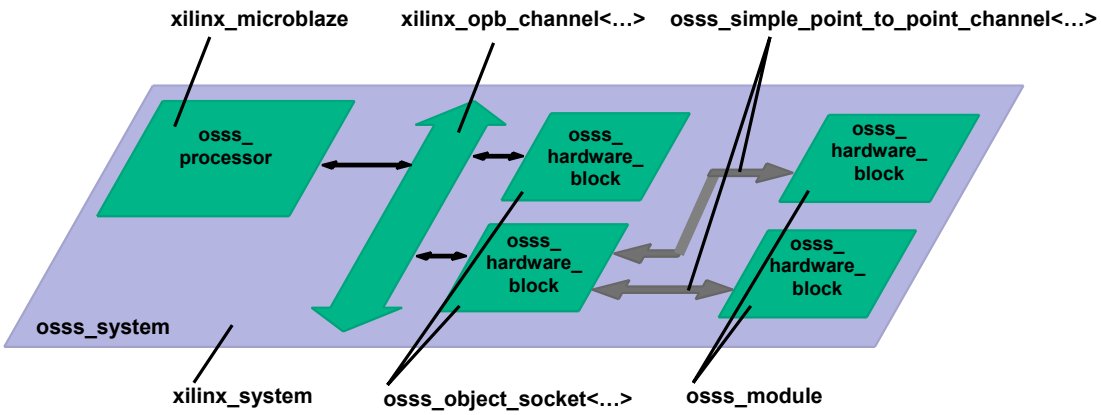



Figure 6.6: Example of a *Virtual Target Architecture* with a single processor and user-defined hardware

Figure 6.6 shows an example of a *Virtual Target Architecture* composed of different *OSSS Architecture Objects*. It includes a single Xilinx MicroBlaze™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.

### 6.2.2 OSSS Device

To model and simulate the target architecture for Recon Objects and Named Contexts, OSSS+R allows the designer to define a device type which provides elements like a reconfiguration controller and information about reconfiguration times.
In most applications multiple Recon Objects will be placed on one FPGA. This will, however, result in shared access to one reconfiguration controller. Therefore, concurrent accesses need to be serialised using a built-in scheduler. OSSS+R provides a default scheduler which can be replaced by a user-defined scheduler.
A significant factor for the overall performance of a reconfigurable system is the time needed for a reconfiguration. To reflect this during simulation, the user can annotate times for the save and restore of attributes and for hardware reconfigurations. Those values are given for every class that is used with a Recon Object or Named Context and are mapped to clock cycles during simulation. The values may be either estimations or back-annotations from synthesis results.

*Note:* The red dotted box around `osss_device` and `osss_system` in Figure 6.5 indicates that these two block should be merged into one in a future OSSS library release.

#### Device Instantiation

OSSS+R distinguishes between device types and device instances. A device type describes features of a device which are identical for all instantiations of that device type such as reconfiguration scheduler and times. All Recon Objects of a design have to be bound to a device.
A device type is created using the `osss_device_type` class. Multiple devices of that type are instantiated using the `osss_device` class. Listing 6.29 shows the instantiation of a device type and a corresponding device instance. Finally, a Recon Object is bound to the device.

```
1 int sc_main(int argc, char* argv[]) {
2   ...
3   osss_device_type my_fpga("MY FPGA");          // device type
4   osss_device my_device(my_fpga,"MY DEVICE"); // device
5   my_module.my_recon(my_device); // bind Recon Object
6   ...
7 }
```

Listing 6.29: Creation of a device

#### Declaring Times

Times are always given for a combination of device type and user class using the `OSSS_DECLARE_TIME` macro. Listing 6.30 demonstrates the definition of times for `my_fpga` and `my_class`.

```
int sc_main(int argc, char* argv[]) {
  ...
  OSSS_DECLARE_TIME(my_fpga, my_class,     // device type + class
                    sc_time(20,SC_US),     // attribute save+restore time
                    sc_time(30,SC_MS));    // reconfiguration time
  ...
}
```

Listing 6.30: Declaration of times

#### Port Bindings

Device objects have clock and reset ports which shall be bound to corresponding signals. These port bindings are either done within a top-level module or within `sc_main` (see Listing 6.31).

```
int sc_main(int argc, char* argv[]) {
  sc_signal<bool> res;
  sc_clock clk("clock",sc_time(10,SC_NS));
  ...
  my_device.clock_port(clk);
  my_device.reset_port(res);
  ...
}
```

Listing 6.31: Port binding of a device.

### 6.2.3 Mapping

The mapping step from the *Application Layer* to the *Virtual Target Architecture Layer* replaces HW components by OSSS modules, maps software tasks onto processors and wraps Shared Objects by so called Object Sockets. A challenging task is the mapping of communication links between software tasks or hardware modules and Shared Objects to appropriate communication resources.

The mapping of HW and SW modules and the mapping of communication links from the Application Layer Model can be performed from two directions:

**top-down:** In a top-down mapping the architecture requirements are dirived directly from the executable application speciification. They are the main driver for finding the "right" architecture. This approach requires a flexible target architecture, especially in the case where the application model should remain unchanged.

**bottom-up:** In the bottom-up approach the target achitecture is more or less fixed. Take for example a System on Chip consisting out of general purpose software processors and DSPs, like the Texas Instruments OMAP (Open Multimedia Application Platform). In this case the appliction model needs to be highly flexible to be mapped onto the pre-defined architecture.

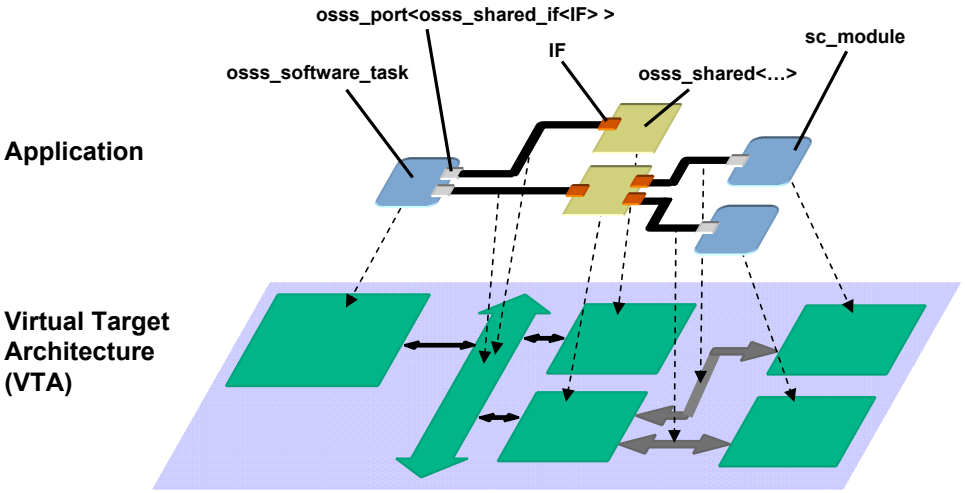



Figure 6.7: OSSS Application to Virtual Target Architecture Mapping (top-down)

However, enabling both approches a flexible Application Layer model and Architecture Layer model is needed. Both approaches have in common that the mapping of abstract communication links on Application Layer to physical communication channels on Virtual Target Architecture is necessary for the overall system implementation as a SoC. In the top-down approach this step is usually called communication refinement. An abstract communication channel is refined to a concrete protocol implementation on a physical channel. For the bottom-up approach the term "communication mapping" seems to be more appropriate since the channel implementation is not derived directly from the application requirements.

The OSSS methodology is targeted on the mapping of an Application to a Virtual Target Architecture without[3] the need for a manual communication refinement. Starting with an application model consisting out of communicating processes using method calls, several manual refinement steps are needed to end up with a synthesisable[4] description of the application. These steps involve

- hardware/software partitioning (which parts of the application or algorithm are implemented in software, and wich parts are implemented in hardware),
- system architecture definition (allocation of architectural resoures like processor types, their number, allocatable hardware resources and memories that can be used for the mapping of the application [top-down], or defintion of a fixed execution plattform, like the TI OMAP [bottom-up]),
- refinement of the hardware and software module behaviour (when starting from an algorithmic description the refinement to dedicated hardware is called behavioural synthesis. It usually performs scheduling, allocation and binding to a constrained set of hardware resources),

[3] or at least with minimal manual effort

[4] In our definition "synthesisable" means, each application component can be further processed by state-of-the-art tools like corss-compilers (for the software part) or RTL synthesis tools (for the hardware part)

- and communication refinement (transferring the abstract (e.g. transaction or method based) communication to a signal based communication channel implementing a defined protocol).

With OSSS we do support the decision of hardware/software partitioning directly. We rather help the designer in finding the "best" solution by manually exploring different application partitionings. The OSSS Application Layer modelling elements (see Section 6.1) can be plugged together easily using Shared Objects as abstract communication medium.

Furthermore, we do not consider behavioural synthesis which convernts algorithmic descriptions into RTL structures. Here we suppose that the behavioural synthesis step has either been performed manually by an external tool. The refinement of the software part is another issue we do not address directly with OSSS. A software task which has been developed and tested on a host system, differnet from the specific target processor, can show unwanted behaviour, when it is directly cross-compiled and executed on the embedded target procesor. To avoid these portability problems we advise the designer to make use of the portable subset of the C++ language [cpp]. Non portable language constructs should be avoided as much as possible.

The main issue we address with the OSSS methodology is communication refinement. Starting from a universal method based hardware/software and hardware/hardware communication we end up with a synthesisable signal based implementation. This implementation can be of different topologies (point-to-point, shared bus and crossbar switch) and protocol implementations.

In the following section we are going to discuss two questions in more detail:

1. How is the mapping from the *Application* to the *Virtual Target Architecture Layer* performed?
2. Starting with a method based communication on the *Application Layer* and ending up with a signal based communication on the *Architecture Layer*. How is this communication refinement performed in a convenient way?

### 6.2.4 Remote Method Invocation

The general meaning of the term *Remote Method Invocation* (RMI) is the call of a method (function or procedure) of an object that is not directly accessible to the caller. I.e. it is not accessible in the instruction memory of the local processor executing certain software. The remote object has to be physically accessible through a communication network. (e.g. Ethernet). On the *Virtual Target Architecture Layer* the communication network between software tasks, hardware modules, and Shared Objects is modelled by OSSS-Channels. Thus, RMI in the context of OSSS represents a method call from a port of a hardware module or software task to an interface of a Shared Object through an OSSS-Channel.

The subsequent sections are organised as follows: We start with the description of the general concept behind RMI. This is followed by the presentation of the OSSS-RMI protocol stack.

Afterwards, we show the particular steps of the OSSS design flow that need to be performed in order to map an application to a specific target architecture. For a better illustration we perform these steps for the simple consumer/producer example from Section 6.1. After the presentation of the mapping steps we demonstrate the flexible communication refinement provided by the OSSS methodology. It enables a simulative architecture exploration, demonstrated with the producer/consumer example. Finally, we are going to have a look at the physical layer of the OSSS-RMI protocol. The OSSS-Channel concept provides building blocks for the specification of synthesisable and cycle accurate communication channels. These channels establish the signal level communication and transform method based into a signal based communication.

### The General Concept

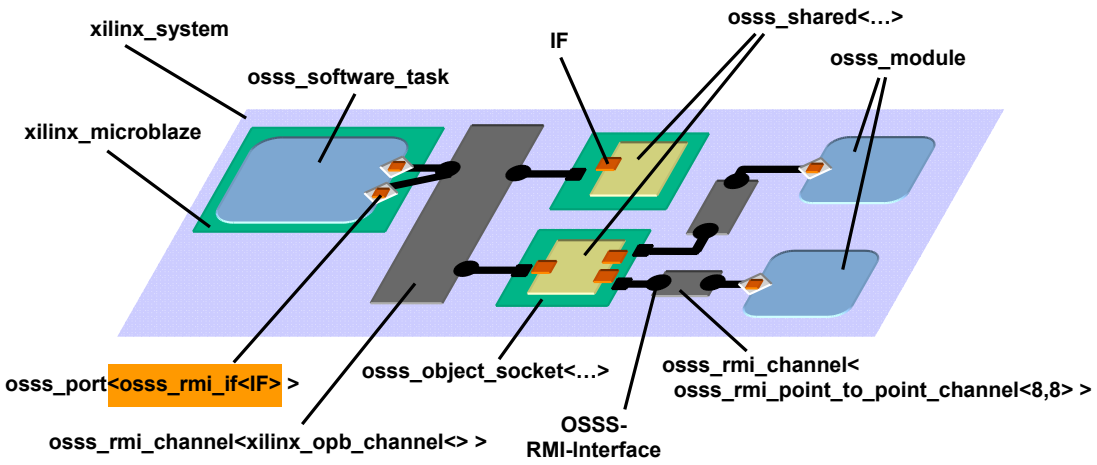



Figure 6.8: Communicating processes mapped on *Virtual Target Architecture*

Figure 6.8 shows the results of the mapping and communication refinement applied to the example presented in Figure 6.7. All communication links from the *Application Layer* have been mapped to `osss_rmi_channel<...>` containers. They serve as wrappers for the OSSS-Channels that implement the physical structure (bus, point-to-point, or crossbar) and the behaviour of the communication protocol. The purpose of the RMI-Channel is the provision of a generic OSSS-RMI interface and the translation of the OSSS-RMI protocol to the generic OSSS-Channel protocol.

The `osss_software_task` has been mapped on a Xilinx MicroBlaze processor that is connected to an OPB, implemented by an OSSS-Channel (`xilinx_opb_channel`). Since we do not want to manually refine all method calls between the Software Task and the Shared Objects, we need to wrap the OPB channel by an OSSS-RMI container (`osss_rmi_channel<...>`). The same wrapping needs to be performed for the point-to-point connection (`osss_rmi_point_to_point_channel<...>`) from the `osss_modules` to the Shared Object.

As already mentioned above, each Shared Object needs to be wrapped by an `osss_object_socket<...>` container. This socket provides a binding mechanism to

the `osss_rmi_channel<...>` container. Moreover, it performs the OSSS-RMI protocol and finally call the remote methods on the Shared Object. Thus serving as a virtual "local client" to the Shared Object.

For calling a remote method from inside of a software task or a hardware module, their communication ports need to be prepared for RMI. This is a rather technical implication. Like in all known RMI concepts a local stub or proxy for accessing the remote object has to be generated. When performing a method call on a local stub the RMI protocol becomes initiated. This implies a sequence of the following operations:

1. serialisation of the remote method arguments, performed at the stub (`osss_rmi_if<IF>`),
2. submission of the client ID, method ID and the serialised arguments, also performed at the stub,
3. reception of the client ID, method ID and the serialised parameters at the remote object socket (`osss_object_socket<...>`),
4. de-serialisation of the parameters at the remote object socket,
5. call of the method by assigning the de-serialised parameters to the method of the remote object (`osss_object_socket<osss_shared<...> >`).

When a method with non-void return parameter has been called, the same protocol is performed on the way back from the remote object to the caller. This caller stub is denoted as `osss_rmi_if<...>`. For RMI communication each OSSS-Port needs to be equipped with this stub. It can be derived automatically from the abstract interface class of the connected Shared Object.

### RMI protocol stack

After the discussion of the mapping process in the previous section, a brief overview of the OSSS-RMI implementation will now be given. Figure 6.9 depicts the RMI protocol stack of the HW/SW interface from Figure 6.18(a). The producer is mapped onto a Xilinx MicroBlaze processor that communicates with the buffer Shared Object by using the RMI protocol over a Xilinx OPB Channel.

When a remote method is called on a local port `osss_port<osss_shared_if<IF> >` ①, the RMI protocol becomes initiated. This implies a sequence of the following operations:

② serialisation of the arguments of the called method (at this point the `serialise()` method is called on each argument of a user-defined class; for all built-in types a predefined serialisation action takes place), ③ the client transactor of the OSSS-RMI-Channel performs the RMI protocol which includes the submission of the caller's client ID, the callee's object ID together with the ID of the called method and the serialised arguments, ④ the master transactor of the Xilinx OPB Channel establishes the signal level communication and

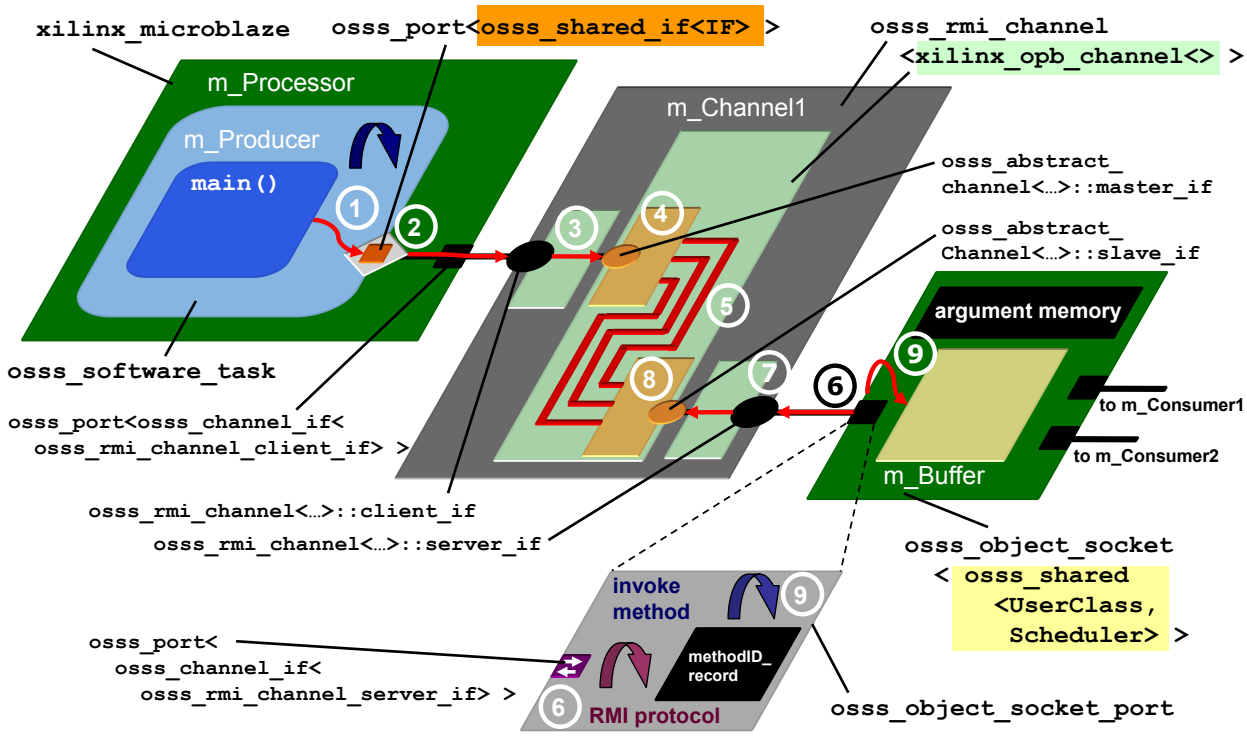



Figure 6.9: Processes communicating through OSSS RMI

transforms the method based communication from the RMI-Channel client transactor into a signal based ⑤ OPB protocol compliant transaction.

The RMI server process inside the object socket port ⑥ performs a method call on the RMI-Channel server interface ⑦ parallel to steps ① - ④. This call is translated by the OPB slave transactor ⑧ to a "listening for action" on the shared bus signals ⑤. When a remote method call to the "listening" object socket is detected the client ID, method ID and the serialised arguments are received by the protocol process ⑥. The serialised method arguments are written to an argument memory inside the object socket. In the last step the `deserialise()` method (as counterpart to `serialise()`) is called on all arguments and the method call on the buffer Shared Object is performed ⑨.

The same protocol has to be performed on the way back from the remote object to the caller, when a method with a non-void return argument has been called. Moreover, this protocol would have been the same for a hardware module instead of a software task as initiator of the communication.

### Relationship to OSSS-Channels

Figure 6.10 demonstrates the repationship between OSSS-Channels and the generic RMI container. OSSS-Channels define the internal structure of the RMI channel. Basically an OSSS-Channel consists out of transactors which translate from a method-based communication to a commnuication on signal level. The method-based communication follow the generic RMI protocol. The OSSS-Channel master transactor translates a aequence of read

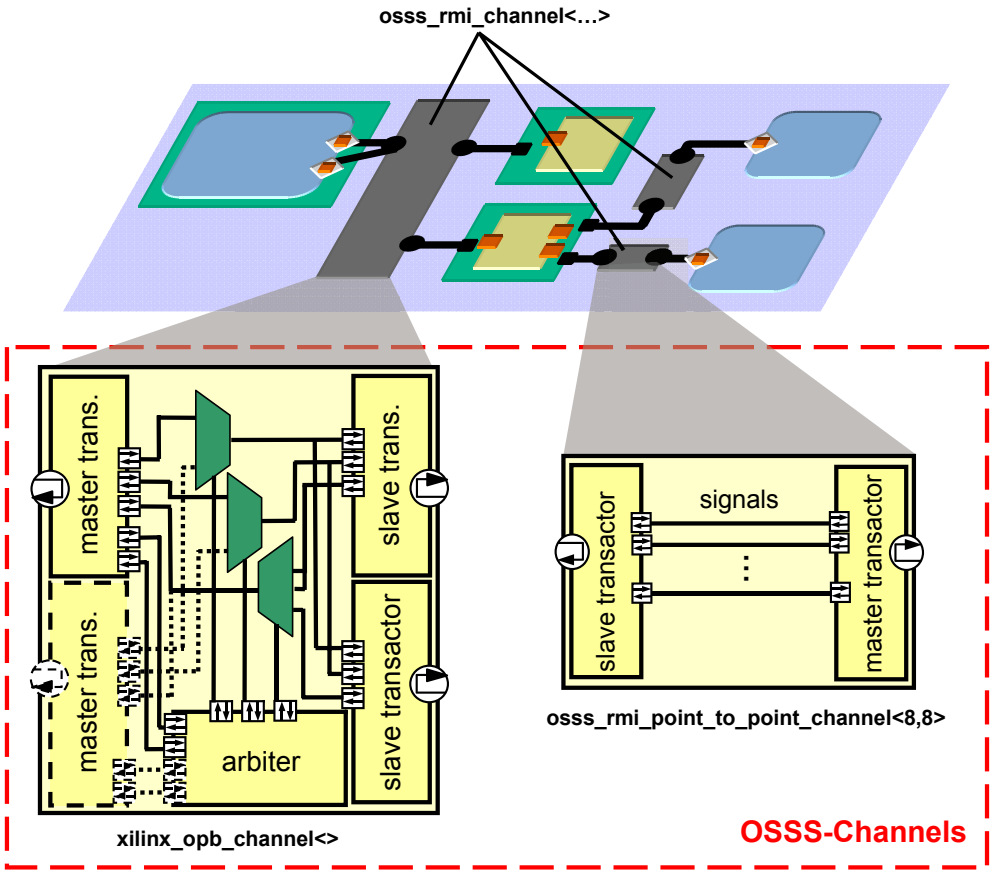



Figure 6.10: Relationship between OSSS-Channels and RMI containers

or write transactions, initiated by through RMI, to a sequence of signal value changes. These signal changes follow a certain communication protocol (e.g. an IBM OPB protocol). The OSSS-Channel slave transactors serve the same purpose for the way back from a certain physical communication protocol to the generic RMI protocol. The signals between the master and slave transactors can be wired in a flexible way to enable different communication topologies, ranging from point-to-point, over shared bus to a cross-bar switch topologie.

If you want to define you own OSSS-Channels or are interested in more details just go on with the next section. All other readers can skip to Section 6.2.6.

### 6.2.5 OSSS-Channels

The OSSS-Channel approach is a concept that enables the designer to model the communication protocol independently from the behaviour of a design. The communication from modules via OSSS-Channel is realised by method calls. The necessary communication primitives are hidden inside of the channel from the calling modules. Hiding communication primitives from the programmer/designer means that the designer of the modules does not have to take care about the explicit implementation of the communication protocol. The module designer only uses the method calls for the data transfer. Building the OSSS-Channel on top of the object oriented features provided by OSSS enables the transfer

of both simple data types (excluding pointers & references) and objects (containing data members of simple and complex type, including class types and array types).

The OSSS-Channels (`xilinx_opb_channel` and `osss_rmi_point_to_point_channel`) are part of the communication library of the *Virtual Target Architecture Layer*. They are used by the RMI concept and encapsulate the Physical Layer of the communication.

The modules connected to an OSSS-Channel can be described at different levels of abstraction always using method calls for communication. The OSSS-Channel has a synthesis semantics and they can easily be integrated in any TLM based design flow.

In the following subsections the OSSS-Channel approach is introduced. It starts with a general overview by presenting the key concepts, followed by a modelling example of a simple point-to-point connection using an OSSS-Channel. After describing these basics, OSSS-Channels which can handle more than one module initiating data transfers are considered. This is done by the example of the `xilinx_opb_channel<...>` that has already been used in the previous sections.

### Key Concepts

The OSSS-Channel concept offers a mechanism to model communication between different modules by method calls. Data transferable by an OSSS-Channel is defined by the hardware/software intersection of OSSS 2.0 (Section 4.4). Unlike channels known from OSCI TLM 1.0 which are always point-to-point connections, our channel approach allows the connection of multiple master and slave modules to a single OSSS-Channel.

The OSSS-Channel concept is based on the separation of port, interface and implementation like it is known from SystemC and encapsulates the user-defined protocols. The separation principle allows the designer to choose from different implementations of a channel as long as they implement the same method interfaces which are used by the modules. This concept allows the separation of application and communication. Additionally, it eases the exploration of different communication protocols for a certain design. This is assisted by storing OSSS-Channels in the library of the *Virtual Target Architecture* for easy reuse.

The OSSS-Channel provides mechanisms to generate the necessary communication infrastructure between the connected modules. The internal structure consists of transactors, arbiters, address decoders and a signal level interconnect network. The generation of the internal structure is invoked by the binding of an OSSS-Port of a module to an OSSS-Channel or by the binding to an RMI-Channel container that encapsulates an OSSS-Channel. The transactors provide a method based interface to the outside of the channel and a signal based interface to the signal network inside the channel. Transactors implement the method interface and utilise the signal based interface to specify a signal level protocol. If more than one master is connected to the channel an arbiter handles the requests. The arbitration mechanism is specified by a user-defined scheduling policy.

The main features of the OSSS-Channel are:

- allows to write user-defined protocols

- allows to connect multiple master and slave modules to the same instance of an OSSS-Channel
- offers a mechanism to model communication between different modules by method calls
- allows to transfer any valid C++/SystemC data types including classes (no pointers & references)
- follows the design principal to separate the port, interface and implementation like it is known from SystemC
- supports the automatic generation/construction of the communication network inside of an OSSS-Channel (invoked by the binding of an OSSS-Port to an OSSS-Channel). This includes the generation of:
  - transactor instances
  - arbiter instances with user-defined schedulers
  - address decoders
  - signal network connecting these channel internals
- allows to create/expand a library of different channel implementations
- has a synthesis semantics

### A Simple Point-To-Point Channel

As an introductive example, we will present an OSSS-Channel implementing a unidirectional point-to-point connection as shown in Figure 6.11. The example consists of two modules (producer and consumer) which communicate directly via an `osss_simple_point_to_point_channel<...>`. The producer initiates write transfers on the channel while the consumer reads from the channel and consumes the transferred data.

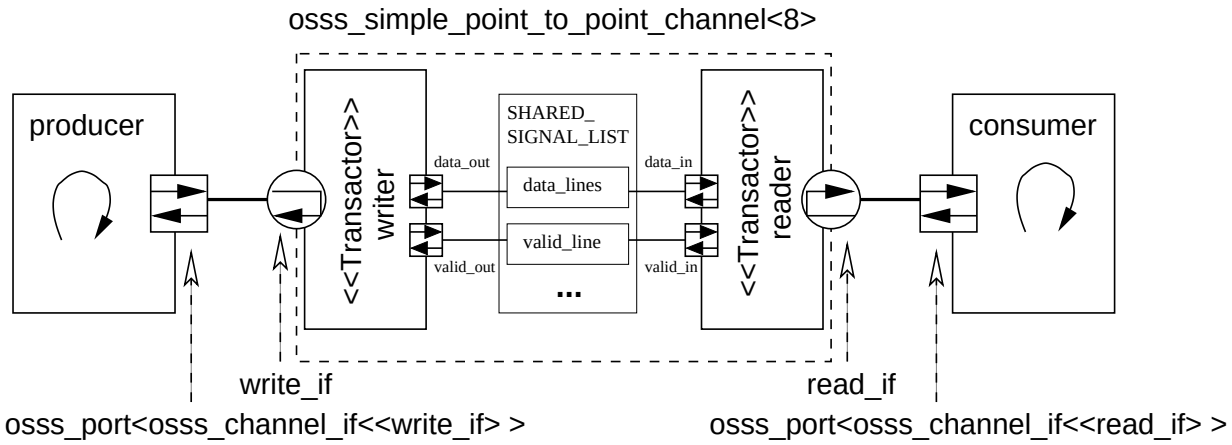



Figure 6.11: OSSS-Channel model of a unidirectional point-to-point connection

Figure 6.12 shows the layer concept of the OSSS-Channel methodology used to implement the `osss_simple_point_to_point_channel<...>`. The presented point-to-point channel implements the `osss_abstract_basic_channel` that contains two inner interface classes, `write_if` and `read_if`. These interfaces define the methods that have to be implemented by the transactors of the channel implementation.

In Figure 6.12 these classes are located at the *Channel Interface Layer*. The design methodology of the OSSS-Channel requires each user-defined channel to implement a certain fixed method interface. Otherwise the separation of application and communication and the exchangeability of channels, providing the same interface but containing different protocol and signal level implementations, is not guaranteed.

The *Channel Implementation Layer* contains the transactors that transform the method based communication initiated from outside the channel to a signal based communication inside the channel.

The bottom layer called *Channel Access Layer* contains the `osss_port_to_channel<...>` port that serves two purposes. Firstly, it creates the channel internals (transactors and their connections to the inner channel signal interface) upon port to interface binding. And secondly, it provides the `operator->()` that is used for calling the methods implemented by the transactors. In the SystemC TLM terminology this kind of module to channel communication is called *Interface Method Call* (IMC) which is often identified as a transaction.

The upper most layer is called *Channel Generation/Service Layer*. It provides services for the signal level connection between transactors inside the channel and is described in more detail later in this section.

In our simple example the `out` and `in` ports of the producer and consumer modules are bound to the point-to-point channel. Each port is declared using the `osss_port<osss_channel_if<...> >` whose template parameter specifies the method interface for accessing the channel (see Listing 6.32). The implementation of the method interface classes is done inside the channels' transactors. Except for the channel internals (transactors and their signal based interconnection) our port to interface binding is compliant to the SystemC port to interface binding methodology. Thus, the point-to-point channel can be used like a `sc_signal` with the possibility to transfer any user-defined data type and still has a well defined synthesis semantics.

```
1  typedef osss_simple_point_to_point_channel<8> Channel_t;
2
3  OSSS_REGISTER_TRANSACTOR(Channel_t::writer, Channel_t::write_if);
4  OSSS_REGISTER_TRANSACTOR(Channel_t::reader, Channel_t::read_if);
5
6  SC_MODULE(producer)
7  {
8    osss_port<osss_channel_if<Channel_t::write_if> > out;
9
10   void producer_proc()
```

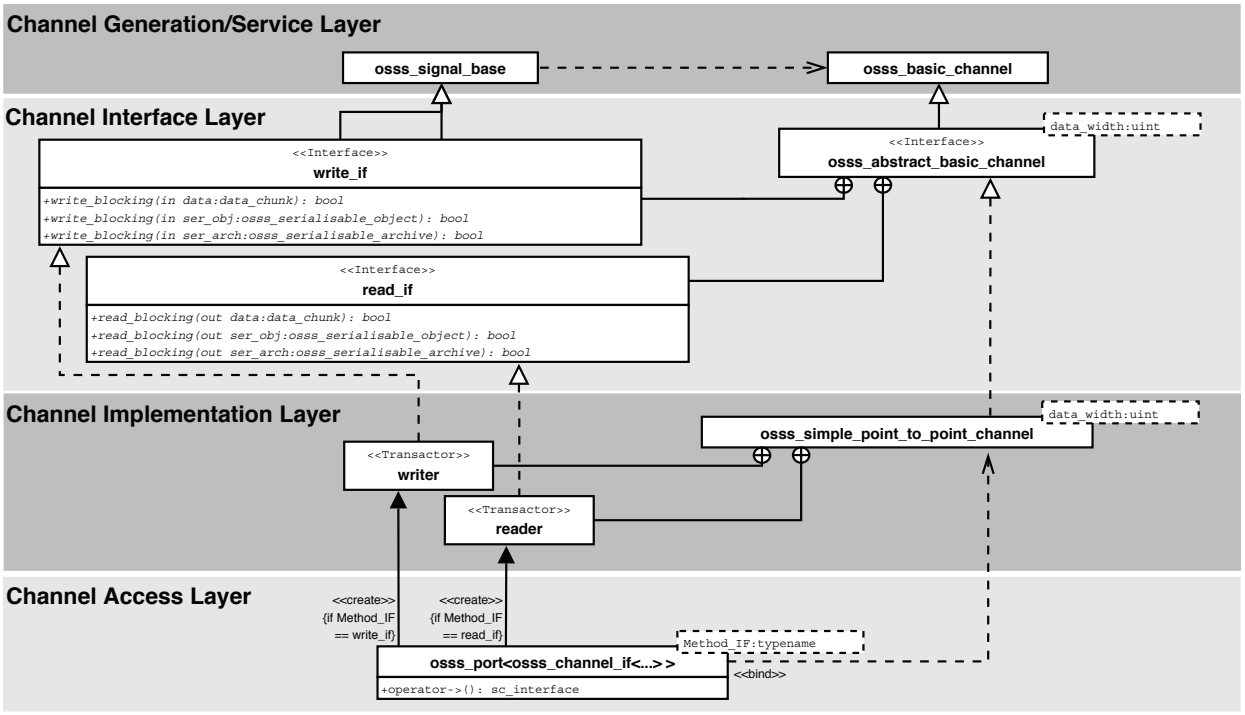



Figure 6.12: Layer concept of the OSSS-Channel methodology used to implement a simple point-to-point channel

```
11    {
12      ...
13      out->write_blocking(...);
14      ...
15    }
16  ...
17  };
18
19  SC_MODULE(consumer)
20  {
21    osss_port<osss_channel_if<Channel_t::read_if>> in;
22
23    void consumer_proc()
24    {
25      ...
26      in->read_blocking(...);
27      ...
28    }
29  ...
30  };
```

Listing 6.32: Point-to-point example *Channel Access Layer*

As already shown in Figure 6.12 the `write_if` requires the implementation of the `write_blocking(...)`[5] method.

[5]Blocking is meant in the sense of calling a method and not returning from that call until its execution has been finished.

This blocking interface method takes an object of type `osss_serialisable_object`. That means each object which should be transferred via an OSSS-Channel has to be derived from the `osss_serialisable_object` class. Thus, each class inheriting from `osss_serialisable_object` has to implement the purely virtual methods `serialise()` and `deserialise()`. These methods declare which members of the class have to be serialised and de-serialised.

The `osss_serialisable_object` class enables the serialisation and de-serialisation of a data object in order to transfer it by the low level protocol via an arbitrary width of the `data_lines` signal inside the channel. The width of the `data_lines` signal has to be determined during declaration of the `osss_simple_point_to_point_channel<...>` by its template parameter[6], see Listing 6.32. This use of the serialisable object base class is necessary to allow the transfer of arbitrary (user-defined) data types by every OSSS-Channel without a modification of the protocol inside of the channel. Thus, always the separation concept of application and communication is guaranteed.

```
1  class writer : public base_type::write_if
2  {
3  public:
4    sc_out< sc_uint< dataWidth > > data_out;
5    sc_out<bool> valid_out;
6
7    OSSS_GENERATE
8    {
9      osss_connect(osss_reg_port(data_out),
10                  osss_shared_signal("data_lines"));
11
12     osss_connect(osss_reg_port(valid_out),
13                  osss_shared_signal("valid_line"));
14   }
15
16   virtual bool write_blocking(osss_serialisable_object& ser_obj)
17   {
18     valid_out = true;
19     wait(2);
20     valid_out = false;
21     ser_obj.serialise_obj();
22     while (!ser_obj.complete())
23     {
24       ser_obj.write_chunk_to_port(data_out);
25       wait();
26     }
27     return true;
28   }
29 };
30
31 class reader : public base_type::read_if
32 {
33 public:
34   sc_in< sc_uint< dataWidth > > data_in;
```

[6] Thus `osss_simple_point_to_point_channel<8>` declares a point-to-point channel with a `data_lines` signal width of 8 bits.

```
35    sc_in<bool> valid_in;
36
37    OSSS_GENERATE
38    {
39      osss_connect(osss_shared_signal("data_lines"),
40                   osss_reg_port(data_in));
41
42      osss_connect(osss_shared_signal("valid_line"),
43                   osss_reg_port(valid_in));
44    }
45
46    virtual bool read_blocking(osss_serialisable_object& ser_obj)
47    {
48      while(valid_in.read() == true)
49        wait();
50      while(valid_in.read() == false)
51        wait();
52      while (!ser_obj.complete())
53      {
54        ser_obj.read_chunk_from_port(data_in);
55        wait();
56      }
57      ser_obj.deserialise_obj();
58      return true;
59    }
60  };
```

Listing 6.33: Point-to-point example *Channel Implementation Layer*

We will now have a closer look at the channel internals, i.e. the implementation of the transactors and their signal based interconnection.

Listing 6.33 shows the implementation of the `writer` and the `reader` transactor of the *Channel Implementation Layer*. The transactors themselves describe the communication protocol by implementing the method interfaces of the *Channel Interface Layer*. The `writer` transactor implements the `write_if` and the `reader` transactor implements the `read_if`. Figure 6.12 shows that both, the write and the read interface inherit from the `osss_signal_base` class of the *Channel Generation/Service Layer*. This base class provides services for the signal based interconnection of the transactors.

Each transactor has an RTL SystemC signal interface consisting out of ports and information about their binding to the signal network inside of the channel. This RTL signal interface is necessary to provide the synthesisable low level port interface which is used to establish the low level communication between the modules connected to the channel. The low level communication protocol itself is encapsulated by the transactors. They implement the `write_blocking(...)` and the `read_blocking(...)` methods by describing how (serialised) objects are transmitted or received using the available low level ports.

In our example a user-defined protocol with two signal level ports (`data_out/in` and `valid_out/in`) for each transactor is chosen. The `valid` port is used to manage the control and the `data` port is used to transfer the serialised object. The width of the data port is determined by the declaration of the channel by its template parameter. These

ports are bound to the channel's communication network inside the `OSSS_GENERATE` section.

In this section the ports of each transactor are connected by the `osss_connect(...)` method. This connection method is used inside the `writer` transactor to connect the `data_out` and the `valid_out` ports to a shared signal called `data_lines` and `valid_line` respectively. It is used the other way round in the `reader` transactor, where connections are defined from the shared signals to the respective ports. In Figure 6.11 this signal based interconnection is visualised by the `SHARED_SIGNAL_LIST` which contains the shared signals `data_lines` and `valid_line`.

The `write_blocking(...)` method generates a control signal that marks the beginning of a data transfer and transmits it via the `valid_out` port. After this notification the serialisable object's attributes are written to a bit vector by invoking the `serialise_obj()` method. This bit vector is divided into chunks of the size of the `data_out` port. The `write_chunk_to_port(...)` method is executed in a loop until the transmission of the whole bit vector is completed.

The `read_blocking(...)` method of the `reader` transactor is the inverse of the `write_bocking(...)` method described above. It awaits the beginning of a transmission and collects the received data chunks until the serialisable object is received completely. In the last stage it rebuilds the serialisable object by assigning the bit vector back to the attributes of the respective object (`deserialise_obj()`).

The `writer` and `reader` transactors inside of the channel are automatically generated and connected to the signal network (shared signals) when a port of the type `osss_port<osss_channel_if<...> >` is bound to the channel. The `OSSS_REGISTER_TRANSACTOR` macro is used to define which transactor has to be generated in dependence of the method interface it implements (see Listing 6.32).

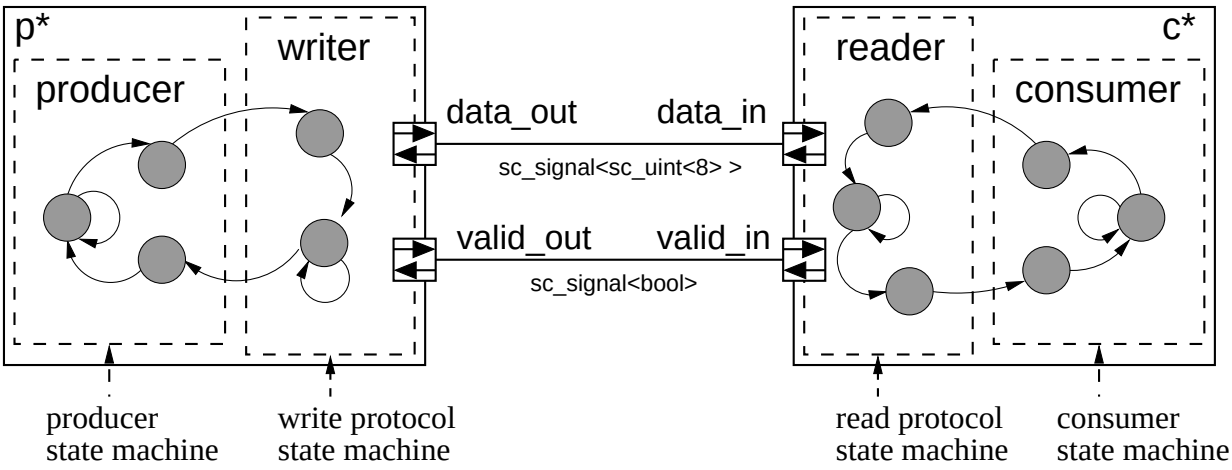



Figure 6.13: Synthesis result of the unidirectional point-to-point channel (`osss_simple_point_to_point_channel<8>`)

The high level synthesis changes the affiliation of the transactors to come to a model which

can be further processed by state-of-the-art RTL synthesis tools. In the modelling phase a transactor belongs to the OSSS-Channel while after high level synthesis it belongs to the connected module. Figure 6.13 shows the synthesis result of the unidirectional point-to-point channel discussed during this subsection. Both `writer` and `reader` transactors are inlined in the producer and consumer module (denoted by p* and c*). More precisely the producer/consumer process results in a state machine that is directly connected with the state machine described in the `writer`/`reader` transactor. The shared signal list is converted into a simple signal based point-to-point connection.

In contrast to an OSCI TLM 1.0 approach, the designer using the OSSS-Channel approach does not need to refine the model which is shown in Figure 6.11 manually to get a synthesisable model. This will be done automatically by a high level synthesis tool called FOSSY (see Section 8.3).

### A Channel with Arbitration

In the previous section an OSSS-Channel constituting a simple point-to-point connection was introduced. Now we consider an OSSS-Channel where multiple master modules use the same transactor type to initiate transactions. Generally, if several processes want to initiate transactions on the same channel, an arbitration mechanism is necessary. This section introduces an OSSS-Channel with a generic arbitration mechanism.

The usage of OSSS-Channels with arbitration is very similar to simple point-to-point connection channels. It uses the same binding, generation and communication mechanisms as discussed previously. Additionally, the user has to distinguish between master and slave method interfaces which provide more generation and connection services for the signal connection inside the channel. This rich set of connection services enables the channel designer to describe different communication architectures like buses or crossbar switches. Besides these services it is possible to integrate user-defined arbiters and user-defined address decoders typically found in state-of-the-art bus implementations.

A multi master/slave OSSS-Channel which is used by several modules can implement different communication protocol schemes defined in its transactors. Each module initiating transactions (master module) on such an OSSS-Channel uses a master transactor. Modules that only react on initiated transactions (slave module) use a slave transactor.

There are two possible approaches to activate a slave module upon a master's request. One relies on a central address decoder activating the passively waiting slaves. The other is to keep the slaves listening to the bus and let them activate themselves when requested (broadcast). We will call the first address decoding centralized address decoding while the second one will be considered as distributed address decoding.

The OSSS-Channel supports both centralized and distributed address decoding. Either, by explicitly specifying an address decoder or by using the predefined `osss_no_address_decoder` dummy, which indicates the use of a distributed address decoding scheme.

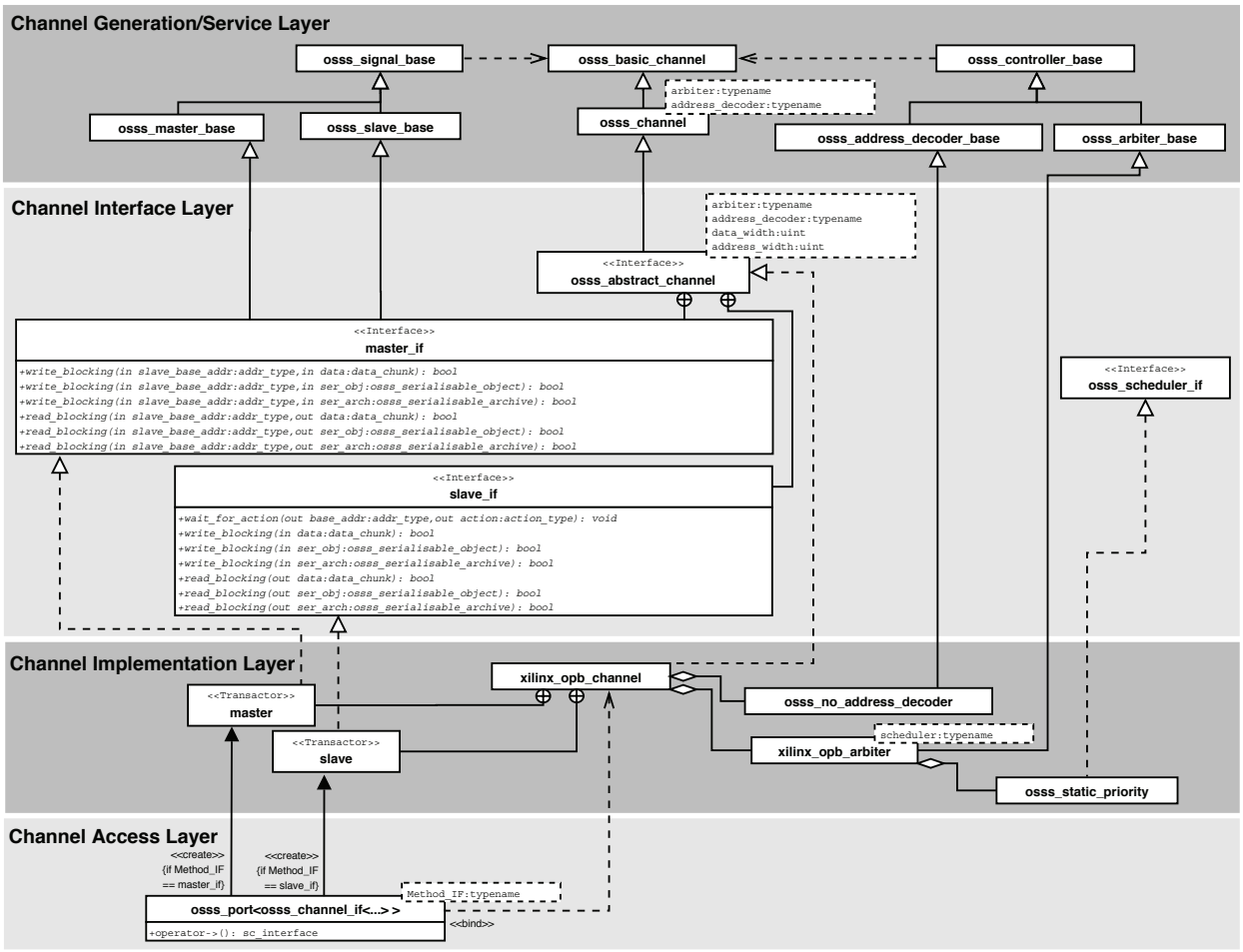



Figure 6.14: Layer concept of the OSSS-Channel methodology used to implement a channel with arbitration

Figure 6.14 shows the software architecture concept used to implement a channel with arbitration. Comparing it to Figure 6.12 it contains additional classes that refine the *Channel Generation/Service Layer*. This refinement is necessary because of the introduced master-/slave communication scheme, the more complex inner channel signal interconnections and the arbiter and address decoder integration. The *Channel Interface Layer* serves the same purpose as in the point-to-point channel. It primary enables the separation of application and communication and secondary the exchangeability of the *Channel Implementation Layer*. The `master_if` declares blocking write and read methods in the same manner as the write and the read interfaces from Figure 6.12. The additional `slave_base_address` parameter is used to specify the destination of the data transfer. Since a slave does not invoke any action on its own, the `slave_if` contains the additional `wait_for_action(...)` method. Called once it waits until the corresponding slave is accessed by a master and returns the base address and the required action (could either be a read- or a write-action) requested by the master. Dependent on the action, the corresponding `write_blocking(...)` or `read_blocking(...)` method of the slave is executed.

Figure 6.15 depicts an extract of the OSSS-Channel simulation model internals using a bus

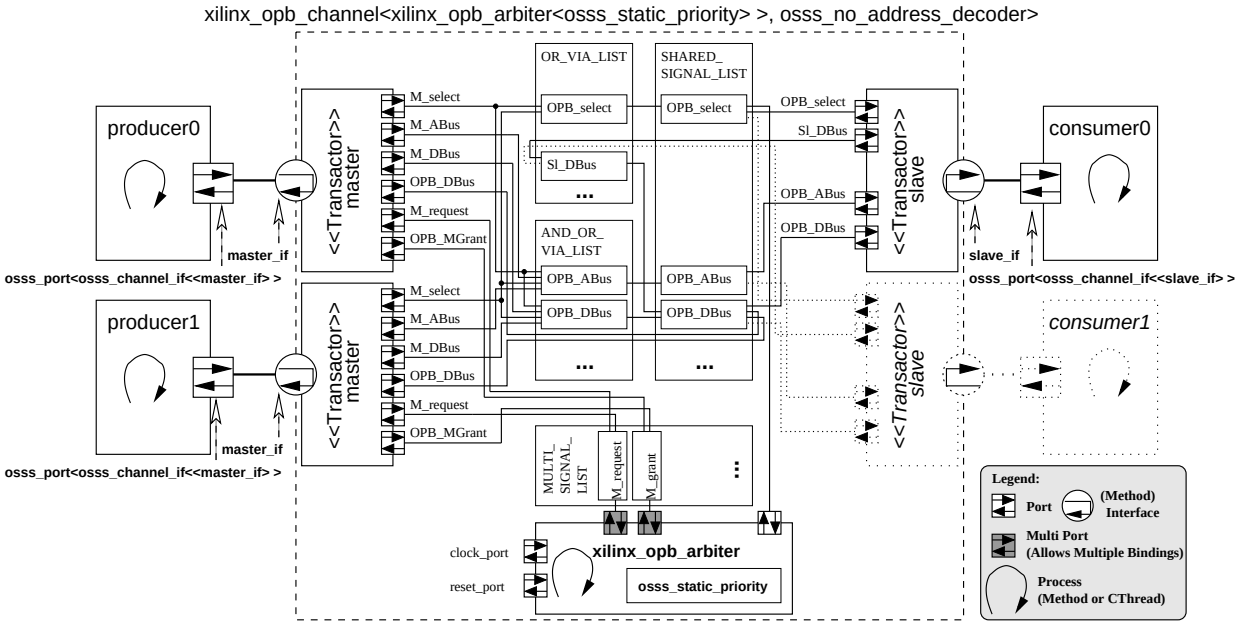



Figure 6.15: The OSSS-Channel model of a channel with arbitration (`xilinx_opb_channel<...>`)

architecture with distributed address decoding as specified in [IBM, Xil05]. It shows three modules connected to different transactors. The `producer0` and `producer1` modules are connected to a master transactor and communicate with it through methods specified in `master_if`. Module `consumer0` is a slave module and is connected to a slave transactor. The low level signal interfaces of the master and the slave transactor are significantly different from each other. Master transactors are not only connected to the shared signals for data and address communication but also to an arbiter component. In addition to the `SHARED_SIGNAL_LIST`, used to connect the `writer` and the `reader` transactor in Figure 6.11, the `OR_VIA_LIST`, `AND_OR_VIA_LIST` and `MULTI_SIGNAL_LIST` have been introduced in Figure 6.15. The `OR_VIA_LIST` contains elements that perform a logical or on its input signals. The `AND_OR_VIA_LIST` contains elements that perform a logical and on a certain pair of input signals and afterwards a logical or on all the outputs of the previous and'ing stage. The `MULTI_SIGNAL_LIST` contains elements that are collecting signals which are not shared but unique to each transactor (e.g. the request and grant signals from the master transactors to the arbitration unit). All these lists and their components are part of the *Channel Generation/Service Layer*. The lists and the components are scalable which gives the flexibility to add an arbitrary number of master or slave transactors to the channel (see `consumer1` and its slave transactor for an illustration what happens if a second slave is attached to the channel).

Another interconnection service provided by the *Channel Generation/Service Layer* is the `osss_mux_via` that models a scalable multiplexer. The user has to specify at least a single output, a single control and one or arbitrary input signals. While the On-Chip Peripheral Bus (OPB) modelled in Figure 6.15 uses the `and_or_via` to implement a distributed multiplexer, the `osss_mux_via` is intended to model a centralised multiplexer.

The `xilinx_opb_arbiter` acts as a special kind of transactor for a user definable scheduling policy. To retain the interchangeability of schedulers, each scheduler that should be used inside an arbiter has to implement the `osss_scheduler_if`. This scheduler interface belongs to the *Channel Interface Layer* (see Figure 6.14).

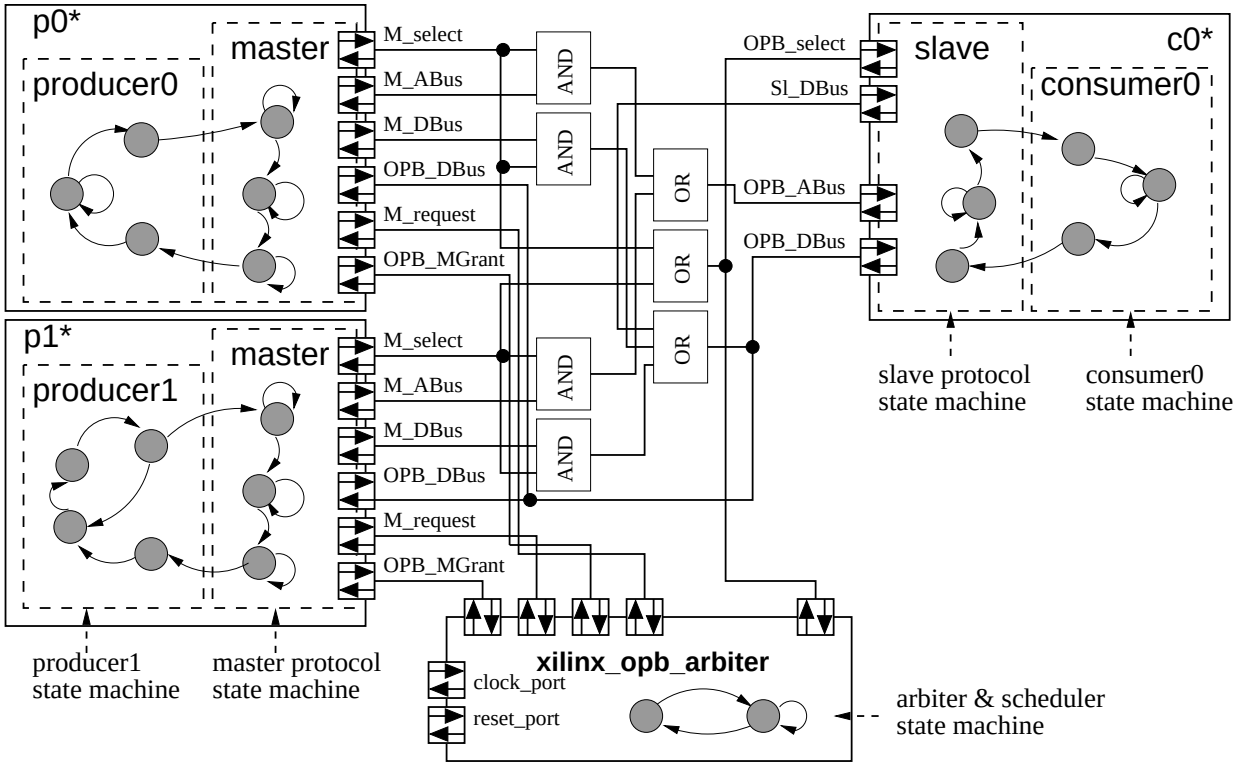



Figure 6.16: Synthesis result of the channel with arbitration

Figure 6.16 shows the high level synthesis result for the OSSS-Channel model of the multi master/slave design. Again each transactor once contained in the `xilinx_opb_channel<...>` is now part of the respective module (denoted by `p0`*, `p1`* and `c0`*). The interconnections inside the channel have been transformed into a communication network consisting of signals, and-gates and or-gates.

Assuming that both, the module and the associated transactor, are described in OSSS 2.0 they form a synthesisable module or entity which is directly connected to the generated interconnection network. Like in the synthesis result of the simple point-to-point connection the master and slave protocol state machines are directly connected to the respective producer/consumer state machines (referred to as method inlining during the synthesis step).

The `xilinx_opb_arbiter` together with its embedded `osss_static_priority` scheduler are synthesised to a single state machine. The inlining of the scheduler to the arbiter state machine works in almost the same manner as the inlining of the producer/consumer with the master/slave state machines.

The multi ports of the arbiter used by the `request` and `grant` signals of the master

transactors have been transformed to single ports. The `clock` and `reset` ports of the producer and consumer module have been omitted in Figure 6.16. But it is important to note, that the `clock` ports of any module connected to the channel must be in the same clock domain (i.e. driven by the same physical clock).

### 6.2.6 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 6.1 to the *Virtual Target Architecture Layer*. The upper part of Figure 6.17 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 6.17 shows the *Virtual Target Architecture Layer*, where the mapping and the communication refinement is accomplished.

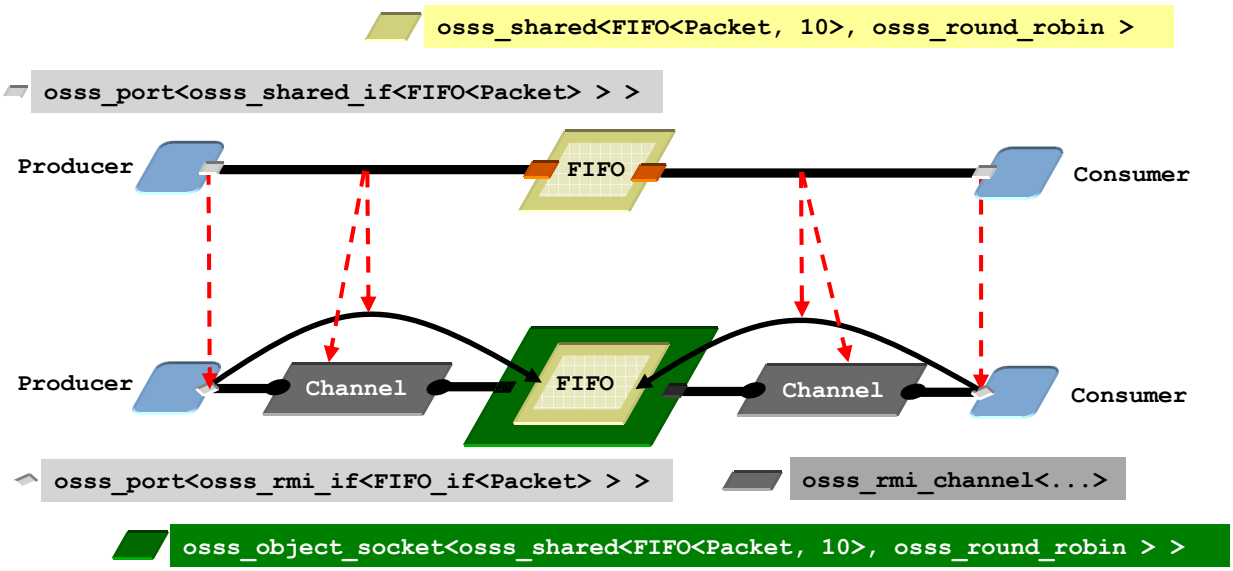



Figure 6.17: Communication refinement of the producer/consumer example

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.

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 6.34 and Listing 6.35) show the replaced ports (changes regarding the *Application Layer* are marked in red color).

```
1  OSSS_SOFTWARE_TASK(Producer)
2  {
3    osss_port<osss_rmi_if<FIFO__put_if<Packet> > > output;
4
5    OSSS_SW_CTOR(Producer) { }
6
7    virtual void main() { ... }
8  };
```

Listing 6.34: Producer with RMI capable port

```
1  OSSS_MODULE(Consumer)
2  {
3    sc_in<bool>    pi_bClk;
4    sc_in<bool>    pi_bReset;
5
6    osss_port<osss_rmi_if<FIFO_get_if<Packet> > > input;
7
8    sc_out<Packet>  po_Packet;
9
10   SC_CTOR(Consumer)
11   {
12     SC_CTHREAD(main, pi_bClk.pos());
13     reset_signal_is(pi_bReset, true);
14   }
15
16   void main();
17 };
```

Listing 6.35: 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 6.36 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 6.36). 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.

```
1  template<class ItemType>
2  class osss_rmi_if<FIFO_put_if<ItemType> > : public FIFO_put_if<ItemType>
3  {
4    public:
5      typedef FIFO_put_if<ItemType> base_type;
6      OSSS_OBJECT_STUB_CTOR(base_type);
7
8      OSSS_METHOD_VOID_STUB(put, OSSS_PARAMS(1, ItemType, item));
9      OSSS_METHOD_STUB(bool, is_empty, OSSS_PARAMS(0));
10     OSSS_METHOD_STUB(bool, is_full, OSSS_PARAMS(0));
11 };
12
13 template<class ItemType>
14 class osss_rmi_if<FIFO_get_if<ItemType> > : public FIFO_get_if<ItemType>
15 {
16   public:
17     typedef FIFO_get_if<ItemType> base_type;
18     OSSS_OBJECT_STUB_CTOR(base_type);
19
20     OSSS_METHOD_STUB(ItemType, get, OSSS_PARAMS(0));
21     OSSS_METHOD_STUB(bool, is_empty, OSSS_PARAMS(0));
```

```
22      OSSS_METHOD_STUB(bool, is_full, OSSS_PARAMS(0));
23  };
```

Listing 6.36: 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 6.37 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.

```
1  class Packet : public osss_serialisable_object
2  {
3  public:
4    OSSS_IS_SERIALISABLE(Packet);
5
6    // default constructor
7    OSSS_SERIALISABLE_CTOR(Packet, ());
8
9    // copy constructor
10   OSSS_SERIALISABLE_CTOR(Packet, (const Packet &pkt));
11
12   // assignment operator
13   void operator=(const Packet &pkt);
14
15   // equality operator
16   bool operator==(const Packet &pkt);
17
18   virtual void serialise() {
19     osss_serialisable_object::store_element(m_source_addr);
20     osss_serialisable_object::store_element(m_target_addr);
21     osss_serialisable_object::store_array(m_payload, 10);
22   }
23
24   virtual void deserialise() {
```

```
25      osss_serialisable_object::restore_element(m_source_addr);
26      osss_serialisable_object::restore_element(m_target_addr);
27      osss_serialisable_object::restore_array(m_payload, 10);
28    }
29
30    unsigned char get_source_addr() const;
31    void          set_source_addr(unsigned char addr);
32    unsigned char get_target_addr() const;
33    void          set_target_addr(unsigned char addr);
34    unsigned char get_payload(unsigned int index) const;
35    void          set_payload(unsigned int index,
36                              unsigned char data);
37    unsigned int  get_payload_size() const;
38
39  protected:
40    unsigned char m_source_addr;
41    unsigned char m_target_addr;
42    unsigned char m_payload[10];
43  };
```

Listing 6.37: 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.

```
1  typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
2    HWSWChannelType;
3
4  typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
5    HWHWChannelType;
```

Listing 6.38: Usage of the `osss_rmi_channel<...>` container

Listing 6.38 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.

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

Listing 6.39: Usage of the `osss_object_socket<...>` container

Listing 6.39 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_module`s by `osss_module`s 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 6.40 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 6.40 can be found in the lower part of Figure 6.17.

In the first part of Listing 6.40 two different kinds of RMI-Channels are defined (q.v. Listing 6.38). 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 6.39).

```
1  #define OSSS_GREEN
2  #include "osss.h"
3
4  class Top : public osss_system
5  {
6  public:
7
8    sc_in<bool>    pi_bClk;
9    sc_in<bool>    pi_bReset;
10
11   typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
12     HWSWChannelType;
13   typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
14     HWHWChannelType;
15
16   typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
```

```
17                                                  osss_round_robin> > BufferType;
18
19    Producer*          m_Producer;
20    HWSWChannelType*   m_Channel1;
21    BufferType*        m_Buffer;
22    HWHWChannelType*   m_Channel2;
23    Consumer*          m_Consumer;
24
25    xilinx_microblaze* m_Processor;
26
27    sc_signal<Packet> ms_Packet;
28
29    Top(sc_core::sc_module_name name) : osss_system(name)
30    {
31      m_Channel1 = new HWSWChannelType("m_Channel1");
32      m_Channel1->clock_port(pi_bClk);
33      m_Channel1->reset_port(pi_bReset);
34
35      m_Channel2 = new HWHWChannelType("m_Channel2");
36      m_Channel2->clock_port(pi_bClk);
37      m_Channel2->reset_port(pi_bReset);
38
39      m_Buffer = new BufferType();
40      m_Buffer->clock_port(pi_bClk);
41      m_Buffer->reset_port(pi_bReset);
42      m_Buffer->bind(*m_Channel1);
43      m_Buffer->bind(*m_Channel2);
44
45      // this is a software task
46      m_Producer = new Producer("m_Producer");
47      m_Producer->clock_port(pi_bClk);
48      m_Producer->reset_port(pi_bReset);
49      m_Producer->output(*m_Buffer);
50
51      m_Processor = new xilinx_microblaze("m_Processor");
52      m_Processor->clock_port(pi_bClk);
53      m_Processor->reset_port(pi_bReset);
54      // this port binds the processor to its bus (m_Channel1)
55      m_Processor->rmi_client_port(*m_Channel1);
56      // here the above software task is added to this processor
57      m_Processor->add_sw_task(m_Producer);
58
59      m_Consumer = new Consumer("m_Consumer");
60      m_Consumer->pi_bClk(pi_bClk);
61      m_Consumer->pi_bReset(pi_bReset);
62      m_Consumer->po_Packet(ms_Packet);
63      m_Consumer->input(*m_Channel2, *m_Buffer);
64    }
65  };
```

Listing 6.40: 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 6.40 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 6.41 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.

```
1 #define OSSS_GREEN
2 #include "osss.h"
3
```

```
4  class Top : public osss_system
5  {
6  public:
7
8    sc_in<bool>    pi_bClk;
9    sc_in<bool>    pi_bReset;
10
11   typedef osss_rmi_channel<xilinx_opb_channel<false, false> >
12     HWSWChannelType;
13   typedef osss_rmi_channel<osss_rmi_point_to_point_channel<8, 8> >
14     HWHWChannelType;
15
16   typedef osss_object_socket<osss_shared<FIFO<Packet, 10>,
17                                          osss_round_robin> > BufferType;
18
19   Producer*          m_Producer;
20   HWSWChannelType*   m_Channel1;
21   BufferType*        m_Buffer;
22   HWHWChannelType*   m_Channel2;
23   Consumer*          m_Consumer;
24
25   xilinx_microblaze* m_Processor;
26
27   sc_signal<Packet> ms_Packet;
28
29   Top(sc_core::sc_module_name name) : osss_system(name)
30   {
31     // register clock and reset ports and make them global
32     osss_global_port_registry::register_clock_port(pi_bClk);
33     osss_global_port_registry::register_reset_port(pi_bReset);
34
35     m_Channel1 = new HWSWChannelType("m_Channel1");
36     m_Channel2 = new HWHWChannelType("m_Channel2");
37
38     m_Buffer = new BufferType();
39     m_Buffer->bind(*m_Channel1);
40     m_Buffer->bind(*m_Channel2);
41
42     // this is a software task
43     m_Producer = new Producer("m_Producer");
44     m_Producer->output(*m_Buffer);
45
46     m_Processor = new xilinx_microblaze("m_Processor");
47     // this port binds the processor to its bus (m_Channel1)
48     m_Processor->rmi_client_port(*m_Channel1);
49     // adds the above software task to this processor
50     m_Processor->add_sw_task(m_Producer);
51
52     //CAUTION: Make shure the Consumer has been chagned from sc_module
53     //         to osss_module. Otherwise automatic clock and reset port
54     //         binding does not work!
55     m_Consumer = new Consumer("m_Consumer");
56     m_Consumer->po_Packet(ms_Packet);
57     m_Consumer->input(*m_Channel2, *m_Buffer);
58   }
59 };
```

Listing 6.41: The top-level module from Listing 6.40 with global clock and reset port bindings

### 6.2.7 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.

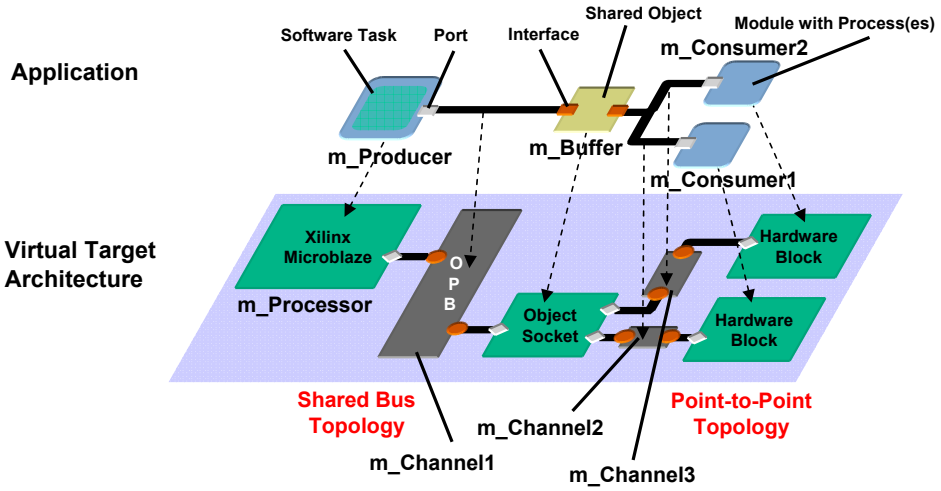
In OSSS communication links (port to interface bindings) are mapped onto communication resources of the *Virtual Target Architecture Layer*, implemented as OSSS-Channels. The provided flexibility for the designer is very high, since these channels can differ in connection topologies (ranging from a point-to-point, over a shared bus to a full featured $N \times N$ crossbar-switch), bit sizes and in their communication protocols. Figure 6.18 illustrates different mapping alternatives of the producer/consumer example introduced in Figure 6.2.

In all mappings shown in Figure 6.18 the producer software task has been mapped to a Xilinx MicroBlaze™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 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.

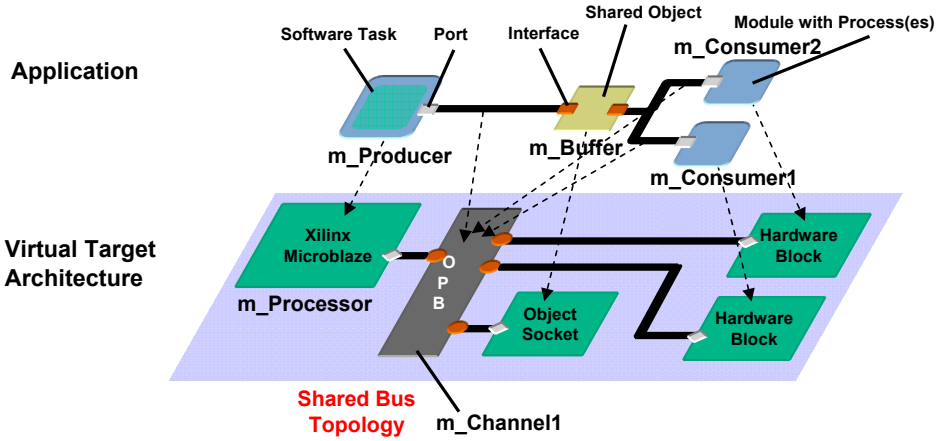
Since the MicroBlaze processor has a built-in OPB interface we have connected it to a Xilinx OPB channel. However, it is also possible to connect any other channel to the MicroBlaze processor, but this would imply the use of a bride or protocol converter. For the sake of simplicity we have chosen the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




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




(b) Single shared bus

Figure 6.18: Different mapping alternatives of the producer/consumer application

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 Section 6.2.4.

Each communication link from the *Application Layer* can be mapped to any `osss_rmi_channel<...>` container (denoted by `m_Channel1 - 3` in Figure 6.18). 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 6.42 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 6.18 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"
6  #include "Consumer.hh"
7
8  class Top : public xilinx_system {
9   public:
10   sc_in<bool> clk, reset;
11
12   typedef
13   osss_rmi_channel<xilinx_opb_channel<> >              Bus_Ch_t;
14
15   typedef
16   osss_rmi_channel<
17     osss_point_to_point_channel<8, 32> >               P2P_Ch_t;
18
19   typedef
20   osss_object_socket<
21    osss_shared<FIFO<Packet, 10>, osss_round_robin> > Buffer_t;
22
23  protected:
24   Bus_Ch_t *m_Channel1;
25   P2P_Ch_t *m_Channel[2];
26
27   Producer *m_Producer;
28   Buffer_t *m_Buffer;
29   Consumer *m_Consumer[2];
30
31   xilinx_microblaze *m_processor;
32
```

```
33  public:
34   Top(sc_module_name name) : xilinx_system(name) {
35     m_Channel1 = new Bus_Ch_t("m_Channel1");
36     m_Channel1->clock_port(clk);
37     m_Channel1->reset_port(reset);
38 #ifdef ALTERNATIVE_A
39     m_Channel[0] = new P2P_Ch_t("m_Channel2");
40     m_Channel[1] = new P2P_Ch_t("m_Channel3");
41     for(unsigned int i=0; i<2; ++i) {
42       m_Channel[i]->clock_port(clk);
43       m_Channel[i]->reset_port(reset);
44     }
45 #endif
46
47     m_Buffer = new Buffer_t("m_Buffer");
48     m_Buffer->clock_port(clk);
49     m_Buffer->reset_port(reset);
50     m_Buffer->bind(*m_Channel1);
51 #ifdef ALTERNATIVE_A
52     m_Buffer->bind(*m_Channel[0]);
53     m_Buffer->bind(*m_Channel[1]);
54 #endif
55
56     m_Producer = new Producer("m_Producer");
57     m_Producer->output(*m_Buffer);
58
59     m_Processor = new xilinx_microblaze("m_Processor");
60     m_Processor->clock_port(clk);
61     m_Processor->reset_port(reset);
62     m_Processor->rmi_client_port(*m_Channel1);
63     m_Processor->add_sw_task(m_Producer);
64
65     m_Consumer[0] = new Consumer("m_Consumer0");
66     m_Consumer[1] = new Consumer("m_Consumer1");
67     for(unsigned int i=0; i<2; ++i) {
68       m_Consumer[i]->clock_port(clk);
69       m_Consumer[i]->reset_port(reset);
70 #ifdef ALTERNATIVE_A
71       m_Consumer[i]->input(*m_Channel[i], *m_Buffer);
72 #else // ALTERNATIVE_B
73       m_Consumer[i]->input(*m_Channel1, *m_Buffer);
74 #endif
75     }
76   }
77 };
```

Listing 6.42: 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 6.2) and the two different Virtual Architecture Models (ref. Figure 6.18). Table 6.3 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 6.3: Simulation results of the different producer/consumer models

| Implementation Model | Simulation Time[a] [s] | Packet Throughput [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.

## 6.3 Summary

### 6.3.1 Passive Modelling Elements

Table 6.4 shows an overview of the passive OSSS *Application Layer* modelling elements available today.

In OSSS we distinguish between value and entity object types.

**Value object** types have an implicit location. They do not describe an own data path, but are embedded in the calling or owning thread. Therefore, member functions of value objects are executed inline to the thread that uses them. Like all objects they are initialisable which usually happens during construction (the constructor is responsible for that). Moreover they are assignable, copyable and serialisable. Assignable means that any other value object of the same type can be assigned. Copyable means initialisation through assignment. Serialisation is a special case of copy that enables to write the state of a value object to a bitvector representation that can either be stored and restored from a memory or can be used to send it through a physical channel, like OSSS-Channels using the OSSS-RMI protocol. Value objects can optionally be polymorphic (denoted by `osss_polymorphic<X>`), which it a form of C++ polymorphism without the use of pointers.

In contrast to value objects **Entity objects** can not be copied and sent around. They have an explicit location with an optionally exclusive data path. Member function calls on entity objects are therefore not inlined in the caller thread. They are executed in their own thread on their own data path. Of course entity objects are initialisable like value objects.

Table 6.4: Overview of passive OSSS modelling elements

| | Plain | Poly<X> | Shared<Y> | Context<X> Recon<Z> |
|---|---|---|---|---|
| Origin | C++ | OSSS | OSSS | OSSS+R |
| Object Type | Value | Value | Entity | Entity |
| Usage | Inline | Inline | Shared | Exclusive/Shared |
| Assign/Copy | Yes | Yes | No (Blue[a]) | Yes |
| Transient | No | No | No | Yes |
| Serialisable | Yes | Yes[b] | No | No |
| Inheritance | Yes | Yes | Blue[c] | Yes |
| Polymorphism | No | Yes | Blue[d] | Yes |
| Arbiter | No | No | Yes (always[e]) | Yes |
| Guards | No | No | Yes | No |
| Locks | No | No | No | Yes (only Recon[f]) |
| Port-IF binding | No | No | Yes | No |
| Requires clock | No | No | Yes | Yes (only Recon[g]) |
| Requires reset | No | No | Yes | Yes (only Recon[g]) |
| Nesting allowed | Yes | Yes | No | No |

with X ∈ { Plain }, Y ∈ { Plain, Poly<X> }, Z ∈ { interfaceof(Plain), `osss_object` }

[a] Only on Application Layer. After mapping and refinement to VTA Layer not possible anymore. We suggest not to use this feature at all.

[b] Not implemented until now.

[c] Not implemented on VTA Layer until now.

[d] Not implemented on VTA Layer until now. Workaround: Use Polymorphic Object as user class of SharedObject.

[e] SharedObjects are always arbitrated. ReconObjects are only arbitrated if required, i.e. when more than a single client is connected.

[f] Locks are only available in ReconObject to explicitly prevent a Context from being disabled.

[g] No clock and reset binding to Contexts is allowed.

Entity objects can be polymorphic as well.

The different object types usually correspond to three different usage patterns: inline, exclusive and shared.

**Inline** means that data and behaviour of an object is embedded into the owning thread. Since this is the strongest from of exclusive (used by a single process) use guards are not allowed. The exclusive usage pattern means that an object is accessed by a single process. This does not require and arbitration and like in the inline pattern guards are not allowed. The main difference between inline and exclusive lies in the structural separation of caller and callee. In the exclusive usage pattern the object has its own dedicated data path apart from the data path of the caller.

The **shared** usage pattern is a special case of the exclusive one (each SharedObject has all properties of an Exclusive Object). In contrast to an exclusive object a shared object is capable of more than a single client. To avoid race condition arbitration through guarded methods (some kind of semaphore) and/or an explicit arbiter is necessary. Table 6.4 shows

that value objects are used inline and entity objects are used exclusive or shared in OSSS & OSSS+R.

In some programming languages (e.g. Java), **transient** is a keyword used as a field modifier. When a field is declared transient, it would not be serialized even if the class to which it belongs is serialized. Until now we do only support transience for state variables in OSSS+R Contexts. State variables that are marked as transient are not preserved when a Context switch on a ReconObject is initiated.

Transience is strongly related to serialisability, whereas **serialisation** is the process of saving an object onto a storage medium (such as a file, or a memory buffer) or to transmit it across a network connection link in binary form. The series of bytes or the format can be used to re-create an object that is identical in its internal state to the original object (actually, a clone). In OSSS plain objects that need to be send through RMI-Channels need to be serialisable. Since OSSS+R does provide another less flexible communication concept, user-defined serialisability for arbitrary object transmissions has not been considered yet. For providing a unified interface concept for OSSS & OSSS+R the general concept of serialisation should be supported by Contexts as well.
As we have described in Section 6.2 communication in OSSS is expressed statically through Port-Interface-Bindings. These binding constitute so-called communication links that are mapped and refined to OSSS-RMI Channels on the VTA Layer.

OSSS+R does not follow exactly the same concept of Port-Interface-Binding. In OSSS+R a method call to a Context is performed directly on the Context itself. There does exist no port which can be bound to the interface of a Context. For a unification of the communication, mapping and refinement concept in OSSS & OSSS+R it is necessary to have the same communication binding mechanism which can be refined to an OSSS-RMI Channel on the VTA Layer. Until now the OSSS simulation library does not support this communication refinement. The proposed modelling style for making use of the flexible communication architecture mapping is the *Mediator Pattern* presented in Section 7.1.

### 6.3.2 Active Modelling Elements

Table 6.5 shows an overview of the active OSSS *Application Layer* modelling elements available today. In contrast to passive modelling elements from the previous section, active modelling elements act as initiators, since they own a thread of control.

In OSSS HW Modules and SW Tasks are the only active modelling elements. A SW Task is a restriced kind of HW Module. The main restrictions are: It contains exactly a single thread and it is not allowed to contain another SW Task (no nesting allowed).

### 6.3.3 Mapping & Refinement in OSSS

Table 6.6 gives an overview of mapping & refinement rules in OSSS. Until now we have a strict separation of HW and SW. HW Modules are refined to synthesisable HW descriptions and SW Tasks are executed on a processor. We are currently working on the support of multitasking. Thus, enabling the mapping of more than a single SW Task to a processor.

Table 6.5: Overview of active OSSS modelling elements

| | HW Module (`sc_module`) | SW Task (`osss_sw_task`) |
|---|---|---|
| Origin | SystemC | OSSS |
| Object Type | Entity | Entity |
| Usage | Active | Active |
| Thread Type | `SC_CTHREAD` & `SC_METHOD` | `SC_CTHREAD` |
| No. of threads | 1 - N | 1 |
| Assign/Copy | No | No |
| Inheritance | Yes | Yes |
| Polymorphism | No | No |
| Requires clock | Yes | Yes[a] |
| Requires reset | Yes | Yes[a] |
| Nesting allowed | Yes | No |

[a] Only on Application Layer. After adding SW Task to CPU no clock and reset is needed. Both clock and reset are taken from the CPU.

Plain Objects can either be used in SW Tasks or in HW Modules. In both cases they become inlined to the thared of control. The state of a Plain Object can be mapped to a dedicated memory. However until now it is not possible to modify attributes of memory mapped objects directly. The support of a Read-Modify-Write mechansim for the purpose of efficient attribute manipulation will be part of a future release of the OSSS library. Polymorphic Objects can be used in the same way as Plain Objects. When using adaptivity in SW we suggest to use the more powerful C++ built-in polymorphism instaed of the Polymorphic Object container class.

Until now Shared Objects can only become dedicated HW. This is performed by wrapping them with a so-called Object Socket. In the context of our works on the support of SW multitasking we think about the support of Shared Objects for inter-task communication as well. The current mapping capabilities of the Recon Object suffer from the same restrictions as the Shared Object. Until now Recon Objects require the availability of dynamic partial reconfigurable HW. Our synthesis technology supports synthesis of static HW from Recon Objects as well, but this is not the preferred use-case. Like with Shared Objects a more flexible mapping concept of Contexts to SW Processors and HW is envisioned.

Communication Links from the OSSS Application Layer (constituted by port to interface bindings) are mapped onto physical communication channels on the OSSS Virtual Target Architecture Layer. With the support of OSSS RMI-Channe containers a generic decoupling from application-specific method-based communication to architecture specific signal-level communication is given. The use of dedicated memories for the mapping of communication links can be used for the implementation of a call-by-reference mechanism.

Table 6.6: Overview of mapping & refinement possibilities in OSSS

| **VTAL Elements** ***AL Elements*** | **Processor** | **HW Block** | **Memory** | **Channel** |
|---|---|---|---|---|
| ***HW Module*** | | change `sc_module` to `osss_module` | | |
| ***SW Task*** | only single Task per processor[a] | | | |
| ***Plain Object*** | inlined | inlined | storage[b] | |
| ***Polymorphic Object*** | use C++ polymorphism[c] | inlined | storage[b] | |
| ***Shared Object*** | Not yet[d] | wrapped by Object Socket | | |
| ***Context & Recon Object*** | | requires dynamic reconf. HW[e] | | |
| ***Communication Link*** | | | Not yet[f] | RMI |

[a] We are currently working on the support of multitasking.

[b] Value objects should be mappable to dedicated memories. Read-Modify-Write semantics should be supported.

[c] It is possible to use a Poymorphic Object in SW, but we suggest to use the more powerful C++ polymorphism instead.

[d] Along with our works on the support of multitasking we think about the support of Shared Objects for inter-task communication.

[e] It is also possible to map/synthesis static hardware.

[f] The use of dedicated memories for the mapping of communication links can be used for the implementation of a call-by-reference mechanism.

# 7 Design Patterns

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* – Christopher Alexander

> *"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context."* – [GHJV95]

This chapter demonstrates how to use the previously introduced modelling elements to solve certain design problems. The Mediator Pattern shows how to establish hardware/software communication with Recon Objects. The Prefetching Pattern demonstrates how to hide reconfiguration delays by loading a new configuration while another one is still in use.

## 7.1 Mediator Pattern

### 7.1.1 Purpose

The *Mediator Pattern* described here can be applied to Application Layer models where Software Tasks need to communicate with Recon Objects. In the currently available OSSS design methodology `osss_software_task`s are not capable to communicate directly with `osss_recon`. This is due to the following restrictions:

- missing Port-Interface binding capability of Recon Objects (since Software Tasks are only allowed to have ports of type `osss_port`),
- and missing communication refinement capabilities. The communication link from a Software Task to a Recon Object needs to be mapped onto the local bus of the chosen target processor executing the Software Task.

For hardware module to Recon Object communication the Mediator Pattern is not applicable, since modules in general are not restricted to have `osss_port`s only. In these scenarios the OSSS+R Recon Object binding concept (`osss_uses`) can be taken (see Section 6.1.7).

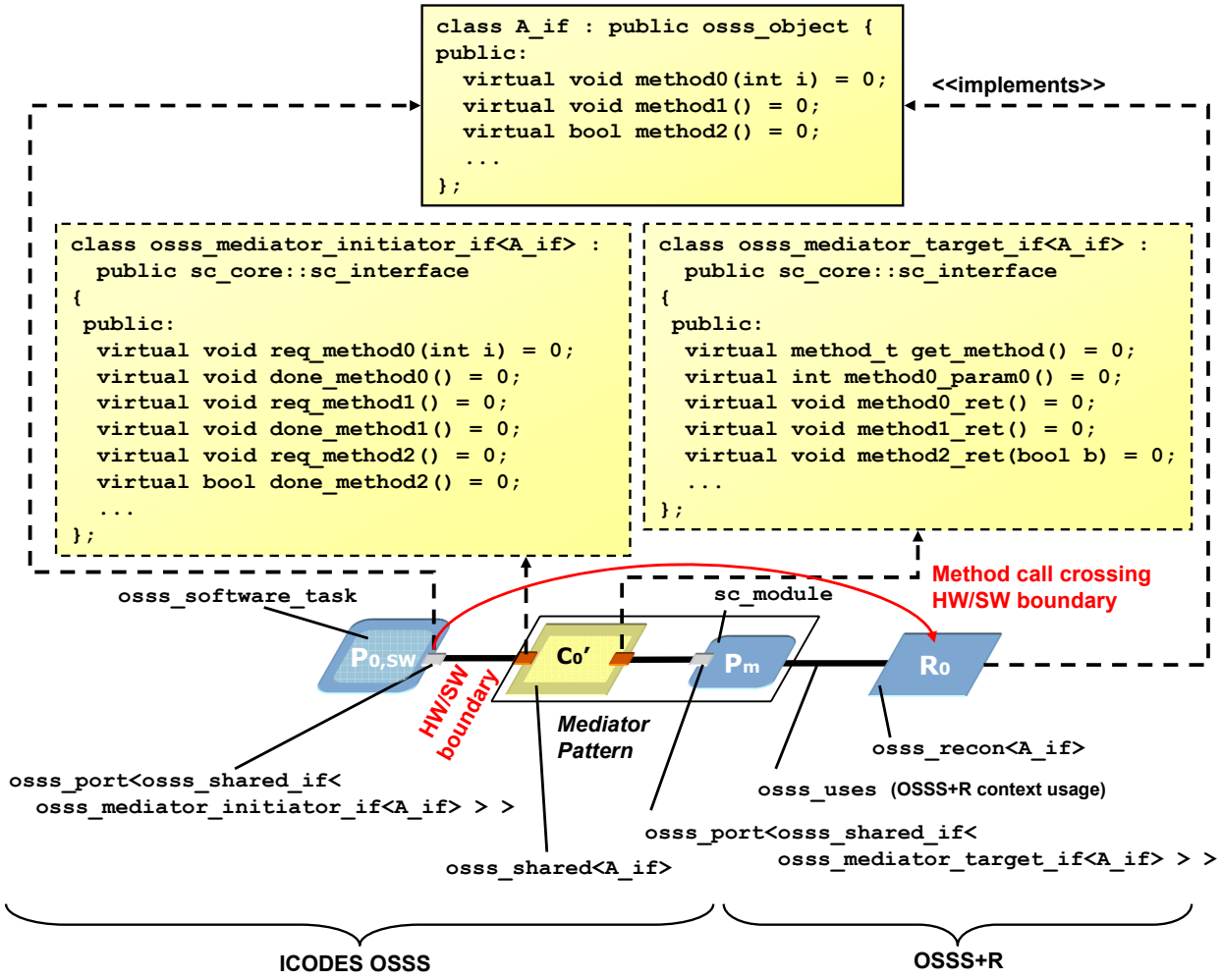



Figure 7.1: HW/SW communication with Context bound to Recon Object

### 7.1.2 Application Layer

Figure 7.1 shows the implementation of the Mediator Pattern on Application Layer. The scenario shows Software Task $P_{0,\mathrm{SW}}$ that wants to communicate with Recon Object $R_0$ through method calls. A direct `osss_port` to Recon Object binding, that crosses the HW/SW boundary, is not possible due to the restrictions mentioned above.
The Mediator Pattern is the preferred way to bridge the gap between hardware and software, retaining method based communication. It consists out of a Shared Object $C'_0$ and Module (with a single process) $P_m$ combination. The Shared Object in this pattern acts as some kind of method call protocol channel, because it breaks down the single transaction from $P_{0,\mathrm{SW}}$ to $R_0$ into a sequence of sub-transactions following a strict protocol sequence.
In the current implementation of the Mediator Pattern we have used the Shared Object directly out of the OSSS simulation library. Just to say it is an "ordinary" Shared Object without any specialisation. Basic properties of the Shared Object used in the Mediator Pattern (called Mediator Shared Object) are:

- It has exactly two clients:

  1. The Software Task calls methods provided by the interface class `A_if` of the

Recon Object on its local OSSS-Port

2. The hardware process calls methods on its local OSSS-Port and translates them to a method call on the Recon Object.

- It implements two interfaces (one for each of the two clients):

  1. `osss_mediator_initiator_if<A_if>` is the interface used by the Software Task. It is specialised for the Context's interface class `A_if` and can be generated automatically.
  2. `osss_mediator_target_if<A_if>` is the interface used by the Hardware process. The hardware side interface has the same specialisation for `A_if` and can also be generated automatically.

- It does not require any arbitration. Due to its special use-case the Mediator Shared Object does not require a scheduler. The guard mechanism provided by Shared Objects suffices to guarantee protocol coherency. In the current implementation a Shared Object always has a scheduler. Therefore, even the Shared Object used in the Mediator Pattern is equipped with a scheduler, although it is not needed explicitly.

- When multiple Software Tasks are accessing the same Recon Object, the above properties require the instantiation of a Mediator Pattern for the connection between each Software Task and the each Recon Object.

The `osss_mediator_initiator_if<A_if>` interface serves for calling methods from the Software Task. It can be generated automatically from the interface class `A_if`. The following methods are generated:

- **Request Class Switch** assigns a new class to the Recon Object.
- **Done Class Switch** confirms that the requested class swicht has been completed.
- For each interface method of `A_if` the following methods are generated:

  1. **Request Method** has the same signature as the method from `A_if` but has a void return type, regardless of the original return type, and it carries the `req_` method name prefix. Its purpose is to request the specified method on the Recon Object.
  2. **Done Method** has the same return type and name that is starting with the `done_` prefix. The argument list is always empty. The purpose of this method it to signal a previously requested method call on a Recon Object has been completed. When the requested method has a return value it is delivered along with the information about method completion.

The `osss_mediator_target_if<A_if>` interface serves for calling methods from the hardware process. Like the software interface it can be generated automatically from `A_if`. The following methods are generated:

- **Method ID Finder** (`get_method`) returns a unique ID of the interface method of `A_if` to be executed on the Context.

- **Class ID Finder** returns a unique ID of the class type that needs to be assigned to the Recon Object.
- For each interface method of `A_if` the following methods are generated:
  1. **Argument Getter Methods** are generated for each argument of the interface method. For interface methods without arguments this method is not generated.
  2. **Return Value Setter Method** is generated for each interface method. The argument of this method has the same type as the return value of the corresponding interface method. For return values of type void the argument of this method is also of type void.

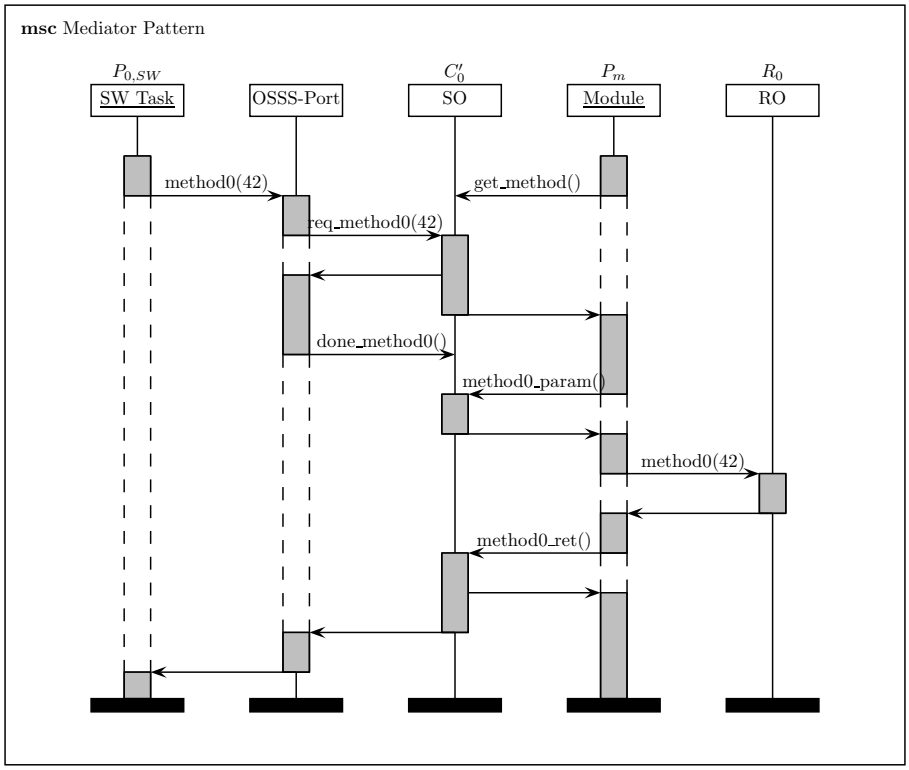



Figure 7.2: Message Sequence Chart of the Mediator Pattern

Figure 7.2 shows the method execution sequence on a Recon Object that is initiated by a Software Task. In the shown message sequence chart the Software Task $P_{0,\mathrm{SW}}$ calls the method `method0(42)` on the Recon Object $R_0$ (cf. Figure 7.1).

### 7.1.3 Mapping and Refinement to VTA

In the previous subsection we have explained the application and functionality of the Mediator Pattern on Application Layer. We now take a closer look on the mapping and refinement of this pattern towards the Virtual Target Architecture Layer.
On Application Layer Figure 7.3 shows the utilisation of the Mediator Pattern from Figure 7.1. The Software Task $P_{0,\mathrm{SW}}$ is assigned to a software processor (e.g. a Xilinx MircoBlaze softcore). Therefore, both communication links $P_{0,\mathrm{SW}} \rightarrow C_1$ and $P_{0,\mathrm{SW}} \rightarrow C'_0$ are

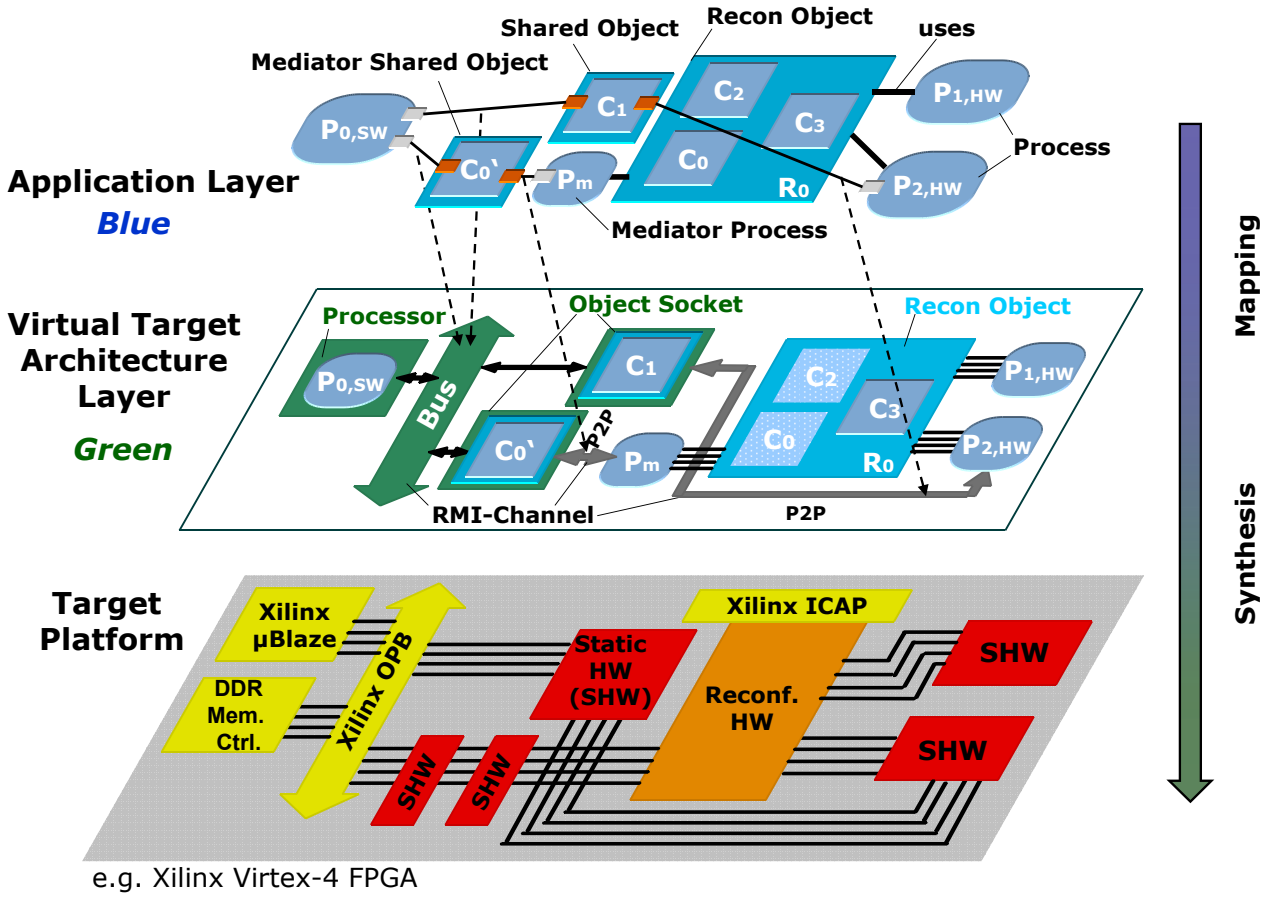



Figure 7.3: Mapping and Refinement flow using the Mediator Pattern

mapped on the same bus (e.g. Xilinx OPB). As discussed in Chapter 6 all channels on the VTA Layer are wrapped by co-called Remote Method Invocation (RMI) Channels. RMI-Channels enable method invocations through physical channels like bus or point-to-point connections. Since the Mediator Shared Object is just an "ordinary" Shared Object it can be easily connected to any RMI-Channel by using the OSSS Object Socket facility. The communication link $P_m \rightarrow C_0'$ is mapped onto a dedicated point-to-point channel. It would have also been possible to map it onto the processor bus.

The communication link $P_{2,\mathrm{HW}} \rightarrow C_1$ is also mapped to a dedicated point-to-point channel using the OSSS communication refinement methodology.

The rest of the "refinement" belongs to the OSSS+R methodology. It encompasses the Recon Object $R_0$ and all communications with it. Since OSSS+R does not really support a separated application and architecture view, the only refinement to be done here is the annotation of architecture dependent reconfiguration and context storage times for Contexts $C_0$, $C_2$ and $C_3$. In contrast to the reset of the OSSS methodology a user-defined communication refinement is not possible. The default Recon Object cross-bar connection to all clients is used.

Considering the OSSS automatic synthesis flow for HW/SW systems, the software processor together with its connected bus is replaced by Xilinx IP components (MicroBlaze, Memory Controller & OPB). The ICODES Shared Object synthesis transforms both Shared Objects $C_0'$ and $C_1$ to static hardware and generates interfaces to the connected communication

resources. The Recon Object is transformed to static and reconfigurable hardware using the OSSS+R synthesis flow. All processes ($P_m$, $P_{1,\mathrm{HW}}$ and $P_{2,\mathrm{HW}}$) are also transformed to static hardware. For modules that are connected to both Shared Object and Recon Object each port is handled by a different interface synthesis: Of course, connections to Recon Objects using Contexts are handled by the OSSS+R interface synthesis. Connections via OSSS-Ports to Shared Objects are handled by the ICODES interface synthesis. Since both kinds of interface synthesis technologies have been implemented in the *Fossy* synthesis tool, this procedure is straightforward.

### 7.1.4 Restrictions

The utilisation of the Mediator Pattern has been chosen in a balance between implementation effort and usability. The implementation of the mediator pattern is rather straight forward, since it builds on top of existing OSSS and OSSS+R modelling elements. Each of these modelling elements has its own synthesis semantics: Shared Objects in conjunction with the OSSS RMI-Channel mapping and interface synthesis for the implementation of HW/SW communication, and Recon Objects as synthesisable Adaptive Objects. By providing macros to generate most parts of the Mediator Pattern automatically the usability is given. Nevertheless, the presented pattern has some small limitations listed below.

**Manual appliance:** The pattern needs to be applied manually on Application Layer models. Most parts of the Mediator Pattern, including the communication protocol, can be generated automatically. This is supported by a set of predefined macros (like `OSSS_MEDIATOR_METHOD_`). But anyhow, the designer needs to apply these macros manually.

**Fragility:** Since the designer needs to take care to correctly apply this pattern it is considered to be more fragile than designing a "pure" ICODES OSSS or OSSS+R model. As one can see from the message sequence chart in Figure 7.2 the high-level communication protocol incorporated with this pattern is not trivial. Possible errors in this communication sequence are hard to debug.

**Decreased simulation performance:** The use of an additional process in each Mediator pattern increases the number of context switches during SystemC simulation. This has a negative impact on the simulation performance and slows down the simulation.

**Increased communication latency:** The Mediator Pattern introduces a layer of indirection between a Software Task and a Context. Since the Mediator Shared Object has a state and the high-level communication protocol takes several clock cycles the communication latency in increased.

**Decreased communication throughput:** The Mediator Pattern high-level communication protocol has a blocking semantics (i.e. the Software Task is blocked until the method call has been completed). This kind of communication does not allow the use of pipelining. Therefore the increased communication latency has direct impact on the communication throughput.

**Limited support of inheritance:** We only support true interface inheritance, i.e. base classes that have no data members and only consist out of pure virtual member functions. This limitation comes from technical restrictions of the OSSS-RMI technology. The used concept is not capable of resolving other than pure virtual methods.

**Increased resource consumption after synthesis:** The insertion of both a Mediator Shared Object and a Mediator Process has impact on the resource consumption in terms of area. This mainly depends on the number of interface methods that need to be handled by this pattern. For large interfaces, consisting of multiple methods with big parameters, the internal state vector of the Mediator Shared Object can grow to a critical size. Even though this effect is caused by the general Shared Object synthesis, we mention it here for the sake of completeness.

## 7.2 Prefetching Pattern

### 7.2.1 General idea

One general approach to improve the overall performance of reconfigurable systems is to hide reconfiguration times by configuration precaching or prefetching techniques. The general idea of this approach is to load a configuration some time before it is actually needed. If it is accessed a switch between two configurations can be done without any delay, completely hiding any reconfiguration overhead to the application layer. In the past this has been investigated in combination with multi-context FPGAs [TCJW97]. This type of FPGA has multiple configuration SRAM cells connected with the same logic block, allowing to hold multiple configuration at the same time and instant switching between these configurations. While this type of FPGAs could not be reprogrammed partially, configuration prefetching could be implementing without considerable additional effort. However these types of FPGAs have never been commercially available, mainly due to their more complicated production process and the higher power consumption in comparison to single-context FPGAs. Recent research has focused on dynamically partially reconfigurable FPGAs [LH02]. Here configuration prefetching can be implemented by distributing the different configurations to two or more reconfigurable areas. While one of the areas is active, unused ones may be reconfigured. In [LCH07] three classes of configuration prefetching algorithms are defined:

1. **Run-time algorithms** use limited knowledge of the current state of the system to guess future configuration requirements. This approach is very flexible and can dynamically adapt to different application scenarios. However it requires additional run-time overhead and may even increase the reconfiguration overhead if wrong decisions are made. It is quite similar to instruction prefetching techniques for software processors.

2. **Complete prediction algorithms** use execution traces or flow graphs to compute optimal or near optimal solutions for configuration prefetching. Although these algorithms usually generate the most efficient solutions, they have a significant computation overhead and can only be used with one application and known input data.

3. **Offline algorithms** use profiling information of an application to generate general prefetching strategies that should fit to most situation that may arise during the runtime of the application. While their solutions are never optimal the computational

overhead during run-time is significantly lower than those of the complete prediction algorithms, making it possible to easily adapt them to different types of application.

All of these prefetching techniques may improve the performance of a reconfigurable system. However the first and third approach cannot guarantee that the reconfiguration time will never have any impact on the application. If a wrong or at least not optimal solution is chosen a certain configuration may no be available if it is accessed, having impact on the system's latency and data integrity. For applications with hard real-time constraints, like signal processing, this might not be acceptable. In those cases a complete prediction algorithm will be necessary.
Currently OSSS+R does not support any prefetching explicitly. However given the current impact of reconfiguration times and the performance gain to be expected from prefetching, we are presenting a design pattern to model a prefetching technique in this section.
The actual prefetching strategy is thereby implemented by the designer and can be chosen depending on the requirements of the application. We decided not to add more transparent via explicit language constructs to OSSS+R, that do not need incorporate application knowledge, since that will result in less efficient implementations. Explicitly modelled prefetching can benefit from application specific properties and might even lead to an efficient offline strategy. Furthermore, the presented design pattern did not require significant changes to the simulation library to reflect the effect of prefetching during simulation.

### 7.2.2 Multi-slot Recon Objects

The first ingredient to the Prefetching Pattern is the optional support of multiple reconfigurable areas (so-called *slots*) under the control of a single Recon Object. This is required to support a wider set of OSSS+R constructs in the prefetching scenario and to orthogonalise prefetching control and the actual computation.
In Figure 7.4, the extended block diagram of a multi-slot Recon Object is shown. Named Contexts – which are now required, since an access to the Anonymous Context would be ambiguous – are still bound to this multi-slot Recon Object as usual. The Access Controller decides, which slot is used for a configuration request and then updates the Crossbar Switch accordingly before granting the permission to the calling process.

```
1  SC_MODULE(my_module)
2    osss_recon< my_class > recon;
3    //...
4    SC_CTOR(my_module) {
5      //...
6      recon.setNumSlots( 2 );                        // enable multiple slots
7      recon.setNumParallelAccessesAllowed( 2 ); //  with concurrent accesses
8    }
9    //...
10 };
```

Listing 7.1: Creation of a multi-slot Recon Object

Obviously, the number of bound Named Contexts has to be larger or equal to the number of slots of the Recon Object. Additionally, the number of parallel accesses can be set, as well. The latter property enables multiple processes to hold a permission to different slots simultaneously. In this case, only the request of a permission on specific contexts is

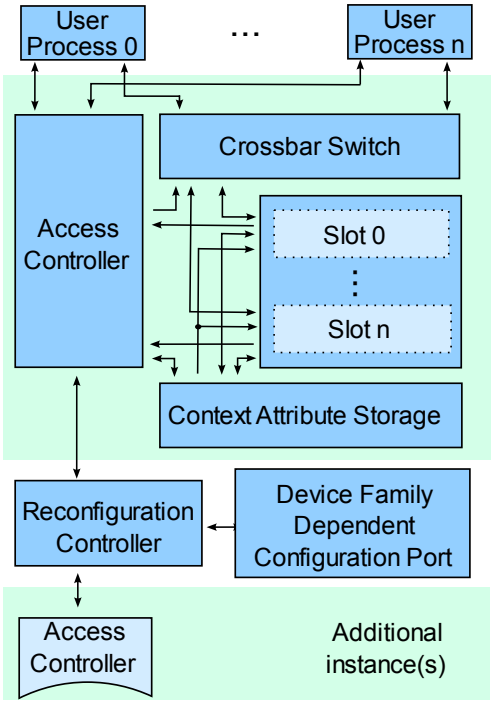



Figure 7.4: Multi-slot architecture of a Recon Object

serialised by the access controller. As will be shown later, both statements are required for the Prefetching Pattern.
Depending on the application, the number of slots (and simultaneous accesses) can of course be adopted to the specific requirements. As a general rule, the number of slots has to exceed the number of client processes, that actually perform the computation, by one. This way, alway one slot can be used for a reconfiguration, while the other slots can still be used for the uninterrupted computation.

### 7.2.3 Using a multi-slot Recon Object for Prefetching

The actual strategy, when to load which configuration is highly application specific, as mentioned before. The most general case of the Prefetching Pattern modelled in OSSS+R consist of the following components:

- A single multi-slot Recon Object $R$ with $n$ slots and $n$ parallel accesses controls the reconfigurable areas.
- At $m \leq n$ client processes $P_1 \dots P_M$ are used for the main computation. Each process $P_i$ uses at least two Contexts $C_{i,1}$, $C_{i,2}$[1], while a switch between the contexts is reasonably seldom to deploy prefetching.
- A *prefetching control* process $PC$ controls the prefetching in parallel for all client processes. $PC$ uses the Recon Object continuously to enable the "next" needed context $C_{i,j}$ in the currently unused slot in an application specific way.

[1] Of course, multiple client processes can share different contexts as usual, but for the sake simplicity this is not explicitly discussed here.

If the configuration changes within the client processes are too frequent to hide the reconfiguration delays for all processes, the number of processes per Recon Object might have to be reduced. In the simplest case, only one client process and one prefetching process are needed. Since the reconfiguration itself is a blocking operation for the triggering process, the separation is necessary. In Figure 7.5, this scenario is depicted.

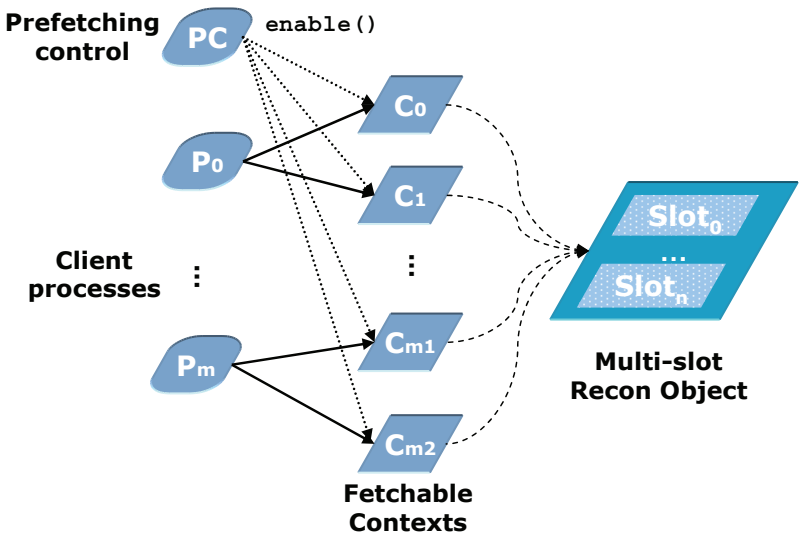



Figure 7.5: Generic prefetching pattern

### Enabling a context

Apart from the actual decision, which Context shall be prefetched to the currently unused slot, the prefetching controller has to actually perform the configuration of said context. This can be done by calling the blocking `enable()` method on the context, which actually requests the permission and immediately returns it. An example of such a prefetching control is shown in Listing 7.2.

```
1  enum context_id { CONTEXT_1, /* ... */ };
2
3  void
4  prefetch_module::prefetch() // prefetching control
5  {
6    // reset block
7    // ...
8    wait();
9    while( true ) {
10     // application-specific prefetching decision
11     context_id next_ctx = select_context_to_fetch();
12
13     switch( next_ctx ) {
14       case CONTEXT_1:
15         // enable chosen context (blocking)
16         context_1.enable();
17         break;
18       // ...
19     }
```

```
20      // application-specific notification of clients
21      // (at least wait until another slot is available)
22      notify_clients_and_sync();
23    }
24  }
```

Listing 7.2: Example of a generic prefetching control

Note, that in the control process, the actual selection of the currently idle slot is not explicitly expressed. There is no way in OSSS+R to forcibly free a slot, or detect which slot is currently not "used". The selection of the to-be-disabled context can be done via an explicit *placer*, which can be set for the multi-slot Recon Object similar to an user-defined scheduler. On the other hand, this imposes the same difficulties as well (see Section 6.1.8). Therefore, we propose an implicit selection through context locking, as described in the next section.
The decision, which context should be fetched next is obviously highly application specific, since this decision most probably depends on the current (or future) input data. Another less obvious application-specific issue is the synchronisation with the client processes. Unless a client process actually uses the freshly prefetched context, the next iteration of the control loop might simply enable yet another context (and thus loose the recently fetched configuration again). Since this is usually not the desired behaviour, Listing 7.2 depicts this requirement in Line 23, calling the imaginary method `notify_clients_and_sync()`.

### Requirements for the client processes

To enable concurrent reconfiguration and computation, it is important that the client processes make use of the `OSSS_KEEP_PERMISSION()` policy (see Section 6.1.8). Since the Access Controller is busy during the actual reconfiguration phase, no new permission can be granted to any client, even if the requested context is currently enabled. This limitation can be overcome by the "lazy" permission handling. In this case, only during the actual context *switch*, the permission has to be returned to the Access Controller. Obtaining the new permission afterwards is then potentially blocking, until the access controller is available again (which again might require synchronisation between the prefetching control and the process, that is about to switch it's currently active context). In an ideal world, the prefetching mechanism never interferes with the clieant's requests. As mentioned in the beginning, this might not be possible to guarantee.
To achieve the required decoupling of the access controller, it is usually sufficient to encapsulate the whole main computation loop in an `OSSS_KEEP_PERMISSION()` block, as shown in Listing 7.3. Here, the previously mentioned synchronisation with the prefetching control process is hidden in the imaginary method `select_context_and_sync()` in Line 13.

```
1  void
2  prefetch_module::compute_1() // client process
3  {
4    // reset block
5    wait();
6
7    OSSS_KEEP_PERMISSION( recon )
8      while( true )
9      {
10       // ...
11
```

```
12        // application-specific context decision
13        context_id current_ctx = select_context_and_sync();
14
15        switch( current_ctx ) {
16          case CONTEXT_1:
17            // use current context (should be enabled :)
18            context_1.do_something();
19            break;
20            // ...
21        }
22        // ...
23      } // end while && keep permission
24  }
```

Listing 7.3: Example of a generic prefetching client

Another important side-effect of the `OSSS_KEEP_ENABLED()` blocks is their impact on the Access Controller's decision, which context is to be disabled due to a reconfiguration request. If all clients have locked their "current" context, the Access Controller automatically selects the only unused slot and thus no further tweaking of the placement decision is required.

# 8 Synthesis

In this part of the manual we will take a closer look at the OSSS synthesis flow and the synthesisable language constructs. We assume that the reader of this chapter is familiar with the OSSS methodology and modelling elements (presented in Chapter 4, Chapter 5, and Chapter 6).

Figure 8.1 shows the synthesis flow for the supported target platform. The upper part shows the tools and libraries developed for processing the OSSS design description while the lower part shows the Xilinx synthesis flow. In the following sections we will sketch the different phases of the synthesis and the connection between the OSSS and the Xilinx synthesis flow.

In general the synthesis flow can be subdivided into an OSSS specific part and a 3$^{rd}$ party backend synthesis flow. Since we have chosen a Xilinx FPGA with a Xilinx MicroBlaze soft processor core the backend flow consists out of the state-of-the-art Xilinx synthesis tool chain. For more information concerning the Xilinx specific tools please refer to [xilb, xilc].

As a precondition we assume that the design has been partitioned into hardware and software parts. The resulting HW/SW design is OSSS 2.2.0 compliant and has been refined from the *Application* to the *Architecture Layer*. This refinement step includes the refinement of each software task, each Shared Object and each hardware module to a description that does not violate the OSSS 2.2.0 synthesis subset. Besides this behaviour refinement the mapping to a specific target architecture has been performed. It basically includes the mapping of a software task to a certain processor and the mapping of the abstract communication links from the *Application Layer* to an OSSS-Channel on the *Architecture Layer*.

To check the functional correctness of this HW/SW design on different levels of abstraction the OSSS 2.2.0 simulation library has to be included. The OSSS 2.2.0 simulation library provides classes for the modelling of hardware and software parts of the system on the *Application Layer*. Additionally, it provides several architecture elements that can be used to assemble the *Virtual Target Architecture*. This *Virtual Target Architecture Library* includes certain processors, memory blocks, sockets for Shared Objects, and OSSS-Channels (see Section 6.2). Besides the structural information provided by these building blocks, the OSSS-Channel enables a clock cycle accurate simulation of the communication on signal level.

The OSSS 2.2.0 simulation library builds on top of SystemC (IEEE 1666-2005 [sys06]).

After a successful simulation of the modelled hardware/software system the synthesis

process can be started. The synthesis flow can be divided into the following phases:

1. OSSS synthesis tools
   (a) Architectural context extraction and hardware/software architecture synthesis (by using the OSSS 2.2.0 Synthesis Library together with an architecture synthesis backend for the *Virtual Target Architecture Library*)
   (b) Software library synthesis (by customizing the OSSS 2.2.0 *Software Library* with information obtained during architectural context extraction)
   (c) High-level synthesis of the user-defined hardware part of the design (by using FOSSY together with the OSSS 2.2.0 *Synthesis Library* constituting a "header-only" version of the SW, HW, RMI and the SystemC Library)
2. Xilinx synthesis tools
   (a) Platform generation using the Xilinx Embedded Development Kit (EDK)
   (b) Low-level synthesis, mapping and place & route (this step can either be performed by the Xilinx Synthesiser Tools or by a third party tool supporting Xilinx devices like Synplify Pro [syn])
   (c) Software library generation for the low-level software drivers for Xilinx specific IP cores by using the EDK (this includes the MicroBlaze soft processor core and the On-Chip Peripheral Bus, OPB)
   (d) Cross-compilation and linking of the software part of the design by using the GNU compiler tool chain for the Xilinx MicroBlaze processor (part of the EDK)
   (e) Bitstream initialisation and downloading to the hardware platform

The synthesis steps concerning the OSSS synthesis tools will be explained in more detail in the following sections.

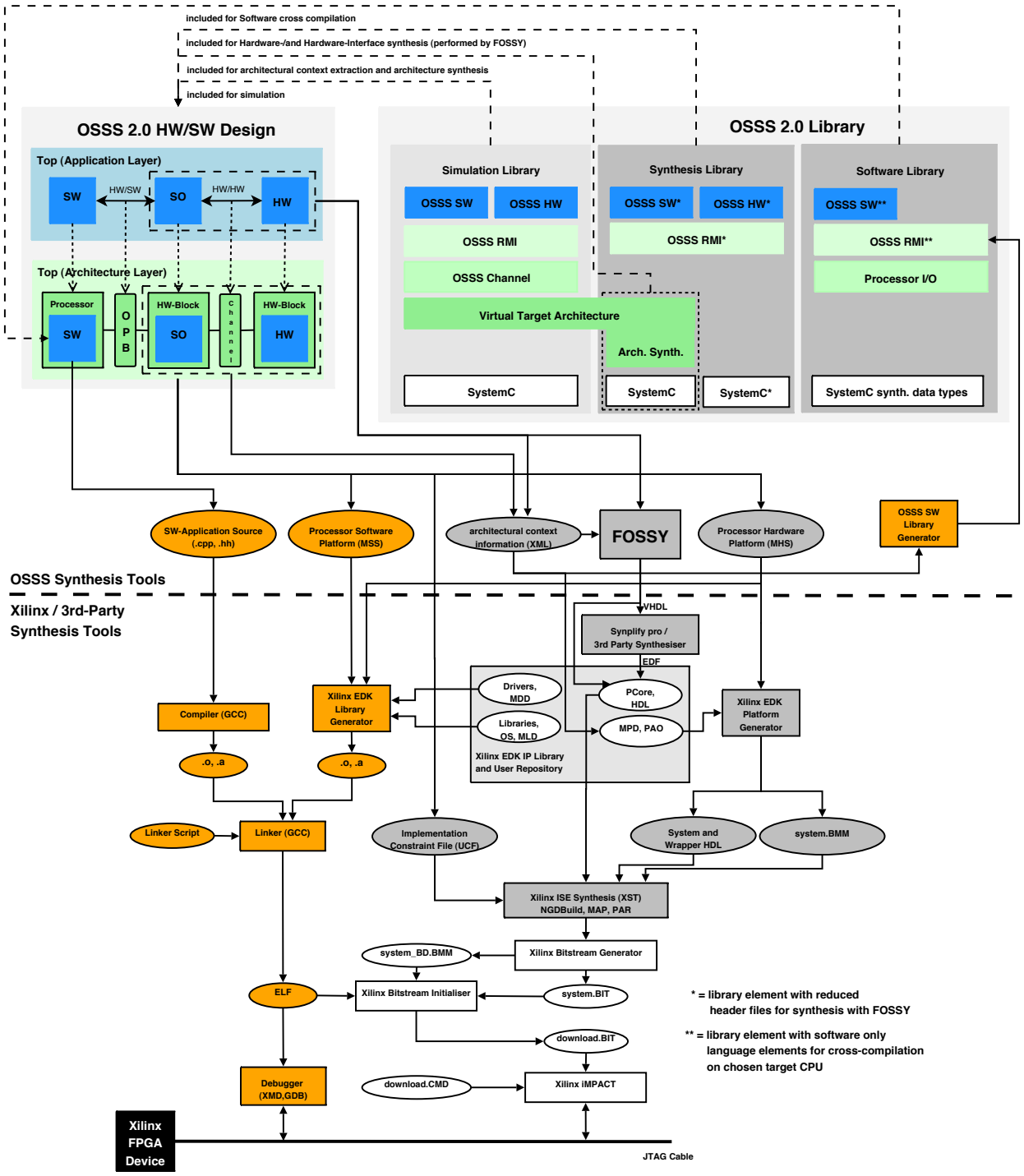



Figure 8.1: The OSSS Synthesis Flow

## 8.1 Architectural Context Extraction and Hardware/Software Architecture Synthesis

The synthesis flow starts with an OSSS hardware/software design which has been mapped from the *Application* to the *Architecture Layer*. Therefore all communication links on the *Application Layer* have been refined by OSSS-Channels, all software tasks are mapped on certain processors, Shared Objects are wrapped by `osss_object_sockets` and all `sc_modules` have become `osss_modules`. That means that all design building blocks are either architecture building blocks from the architecture class library or wrapped by them.

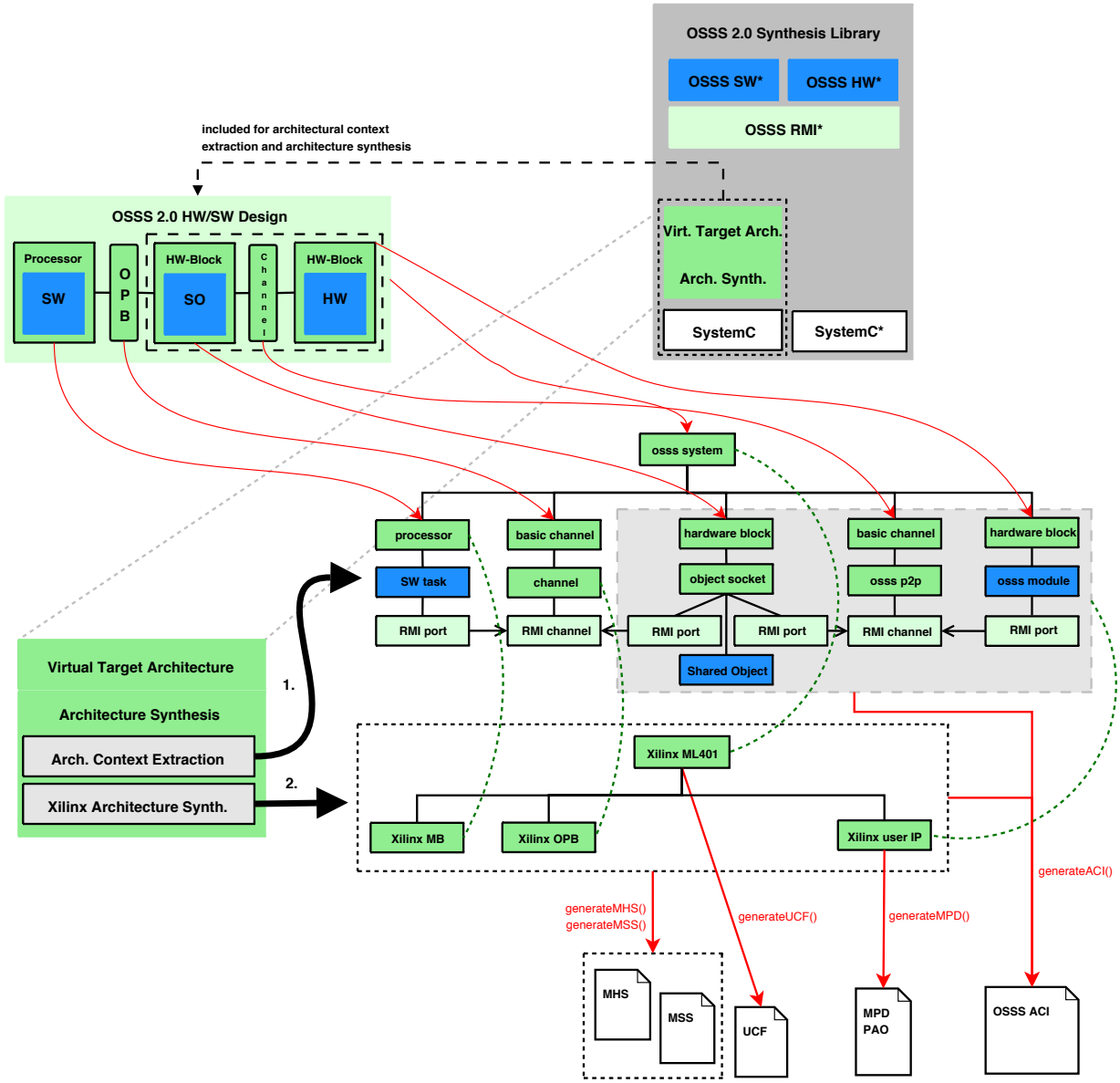



Figure 8.2: Visualisation of the architectural context extraction

Figure 8.2 visualises the architectural context extraction and hardware/software architecture synthesis for the chosen prototyping board Xilinx ML401.

In the first step of the synthesis flow the OSSS HW/SW design is elaborated and architectural context information used for further synthesis steps is extracted. This is done by the architecture synthesis part of the OSSS synthesis library that utilises the SystemC kernel to extract the design structure. The necessary information is extracted at the end of elaboration phase of SystemC. After extracting and analysing the structure of the design the following intermediate data is generated:

1. Architectural context information stored in an XML format that can be used by the FOSSY high-level-synthesis tool and the OSSS software library generator
2. Microprocessor Hardware Specification (MHS) used by the Xilinx Embedded Development Kit (EDK)
3. Microprocessor Software Specification (MSS) used by the Xilinx EDK
4. User constraint file (UCF) used by the Xilinx Integrated Software Environment (ISE)

The MHS and the MSS are both Xilinx EDK proprietary formats. The UCF is Xilinx ISE proprietary and the architectural context information XML file is FOSSY proprietary.

### 8.1.1 Architectural Context Information

The idea of the architectural context extraction is based on a structure reflection during runtime. This is possible because most of the design structure information is inherent in the simulation context which is built and managed by the SystemC kernel. During the elaboration phase which is part of the execution of a SystemC application the module hierarchy is built up. SystemC provides a convenient programming interface to access this module hierarchy.

The `end_of_elaboration()` method is part of the standard implementation of the classes `sc_module`, `sc_port`, `sc_export` and `sc_prim_channel`. By default these functions do nothing. They need to be overridden in a user-defined subclass of these classes in order to perform structure reflection. At the end of the elaboration phase no further modifications are allowed to the module hierarchy. Therefore, it is reasonable to use this hook to start the structure reflection.

```
1  virtual void end_of_elaboration() {
2    std::vector<sc_object*> tops = sc_get_top_level_objects();
3    recursive_descent(tops);
4  }
5
6  void recursive_descent(std::vector<sc_object*>& obj_vec) {
7    for(unsigned int i=0; i<obj_vec.size(); i++) {
8      if(sc_module* module = dynamic_cast <sc_module*>(obj_vec[i])) {
9        sc_interface* channel_if = dynamic_cast<sc_interface*>(obj_vec[i]);
10       if (channel_if) {
11         //add hierarchical channel to design hierarchy
12       }
13       else {
14         //add module to design hierarchy
15       }
16       std::vector<sc_object*> children = module->get_child_objects();
17       recursive_descent(children);
18     }
19     else if (sc_port_base* port =
20                dynamic_cast <sc_port_base*>(obj_vec[i])) {
21       //add port to design hierarchy
22     }
23     else if(sc_prim_channel* channel =
24               dynamic_cast<sc_prim_channel*>(obj_vec[i])) {
25       //add primitive channel to design hierarchy
26     }
27     else if (...) {
28       ...
29     } // fi
30   } // rof
31 }
```

Listing 8.1: Example of structure reflection using the SystemC library

Listing 8.1 gives an example for the usage of the `end_of_elaboration()` method and illustrates how the structure of a SystemC can be traversed. The function `recursive_descent(...)` gets a vector of `sc_object` pointers. The `sc_object` is a

base class of each structural design element in SystemC. A module or a port of a module is a specialisation of that base class. Thus the `dynamic_cast<...>(...)` operator can be used during runtime to check whether a `sc_object` is a module, a port or a primitive channel.

In addition to the module hierarchy information, the port to interface binding can be extracted during runtime. In this context a detailed knowledge about interfaces (`sc_interface`) is desirable. Since interfaces are not derived from the base class `sc_object` and therefore do neither belong to the module hierarchy nor to the object hierarchy it is more difficult to extract them. By making use of the `typeid` operator of the C++ runtime type information (RTTI) it is possible to get the type name of each port.

This structure reflection mechanism has been extended in order to extract the architectural context information in the OSSS synthesis flow.

Figure 6.5 shows the architecture class library which contains the supported architecture building blocks. This library contains general elements that are independent from a certain target platform. These are the `osss_architecture_object` base class and the `osss_processor`, `osss_hardware_block`, `osss_memory` and the `osss_basic_channel`. These classes can be considered as base classes that categorise architecture elements and serve for the extension by more target platform specific elements. Up to now the OSSS Synthesis Library only contains platform dependent building blocks which are IP cores from Xilinx. These are the MicroBlaze soft processor core, Block RAM, external Memory and the On-Chip Peripheral Bus (OPB). The concept of the architecture class library allows also supporting other target platforms like for instance from Altera. In this case the library has to be extended by the IP cores provided by Altera.

As one can see from Figure 6.5 the `osss_architecture_object` is derived from the `sc_object` class. Moreover all OSSS ports are derived from the appropriate SystemC port class `sc_port<IF, N>` and all interfaces are derived from the `sc_interface` base class. Hence it becomes possible to use the reflective approach that has been sketched in Listing 8.1.

As shown in Figure 8.2 the structure reflection starts with the architectural context extraction at the end of elaboration phase. During this phase a tree that represents the structural hierarchy of the OSSS HW/SW design is generated. In Figure 8.2 the architecture of a simple hardware/software design is extracted. The top level module describes an OSSS system that defines the system boundary and connections to the "external world" like clock and reset ports. This system contains a processor that executes a software task with an RMI port that is connected to an RMI channel. On the other hand, the system contains a hardware block that is an object socket that contains a Shared Object and two RMI ports. One of the RMI ports is connected to the same RMI channel as the processor and the other RMI port is connected to another RMI channel that represents a point-to-point (OSSS p2p) connection. The last architecture element is a hardware block which is an OSSS module that contains an RMI port that is connected to the point-to-point as well.

After the structure reflection has been completed the top-level design (which needs to be derived from `osss_system`) is partitioned. There are three kinds of partitions:

- **Software Subsystem:** A software subsystem conatins at least a single processor, its associated software tasks and the bus to which the processor is connected. This partition is considered as an IP block, since both the processor and the bus are taken from an existing IP library of the Xilinx EDK.

- **Custom IP:** A custom IP contains any number of hardware modules and Shared Objects. The main restriction on custom IP block is, that they are only allowed to contain point-to-point channels. This is an important restriction, because all custom IPs that are identified in this synthesis step are further processed by FOSSY. Until this time FOSSY is not capable of performing a true channel synthesis. The only channels FOSSY is able to process are point-to-point channels. But this restriction is rather technically because buses like the Xilinx OPB are provided as channel IPs by the Xilinx EDK.

- **Channel IP:** All channels which are not connected to a software processor belong into this kind of partition. A channel IP conatins only a single channel. In the current state of the OSSS synthesis flow this can only be a Xilinx OPB.

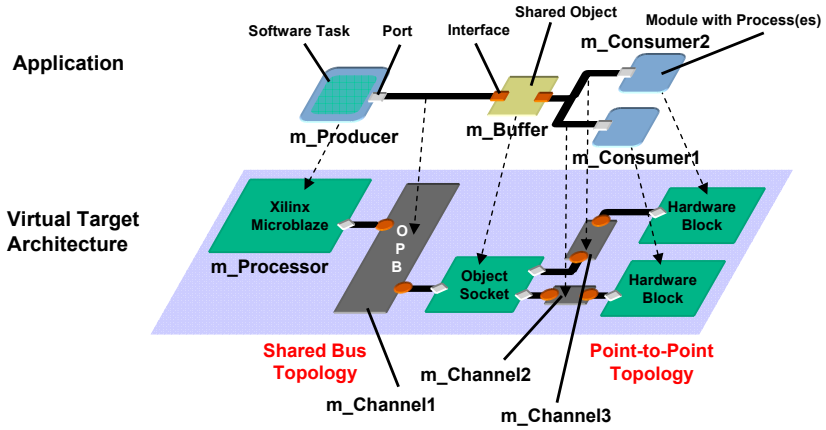
Figure 8.3 shows the result of the system partitioning based on the producer/consumer example mapping alternative A. The software subsystem `osss_software_subsystem_0` contains the producer software task that is mapped on a processor of type Xilinx Microblaze and its associated Xilinx OPB channel. The custom IP `osss_custom_ip_0` contains both consumer hardware modules, the buffer Shared Object and the two point-to-point channels.

The next Figure 8.4 shows the result of the system partitioning based on the second mapping alternative of the producer/consumer example. The software subsystem `osss_software_subsystem_0` stays unchanged compared with Figure 8.3. Since both point-to-point channels from the above example have been replaced by a single Xilinx OPB channel we now have to deal with three instead of a single custom IP. The partitioning results in a custom IP for each consumer hardware module and the buffer Shared Object since all of these components are connected to the second Xilinx OPB channel. The channel itself is encapsulated by the `osss_channel_ip_0` partition.

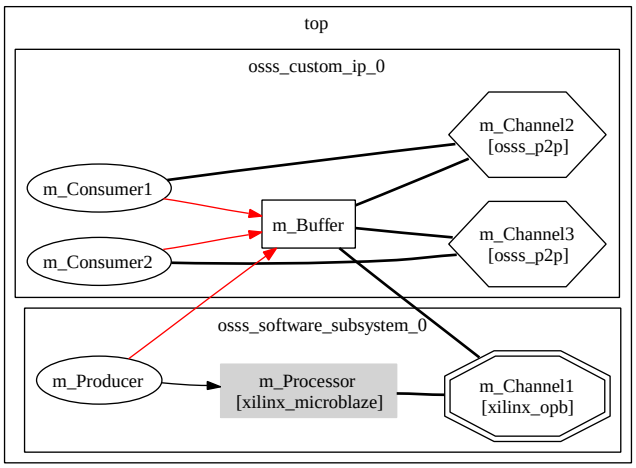
The last Figure 8.5 shows the result of the system partitioning based on the third mapping alternative of the producer/consumer example. Compared to Figure 8.4 the only difference is the missing Xilinx OPB channel `m_Channel2` because all communication links from the *Application Layer* have been mapped on `m_Channel1`.

The step of the architectural context extraction is followed by the target platform specific architecture synthesis. This is done by looking at the most specialised target specific class of each architecture element identified during architecture context extraction.

Considering the example from Figure 8.2 the OSSS system is specialised by the Xilinx ML401 development board. This includes the specific ports from the Xilinx Virtex 4 FPGA to the devices mounted on the ML401 board. This includes the input clock, reset, external memory and I/O pins. The processor is specialised by a Xilinx MicroBlaze that is




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



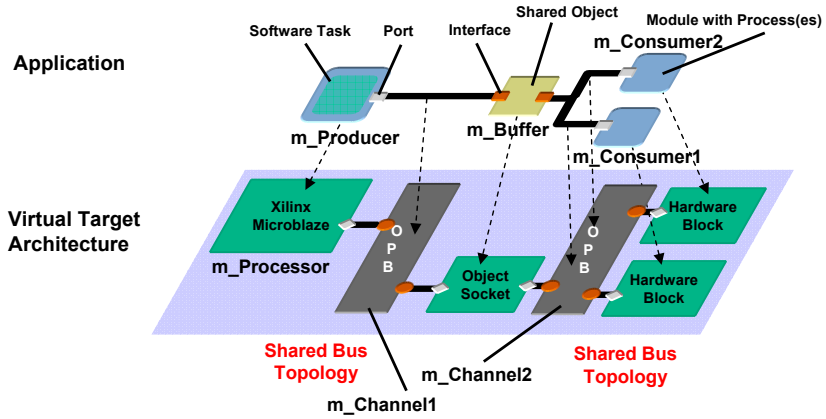

(b) Partitioning view

Figure 8.3: Partitioning of the producer/consumer example mapping alternative A

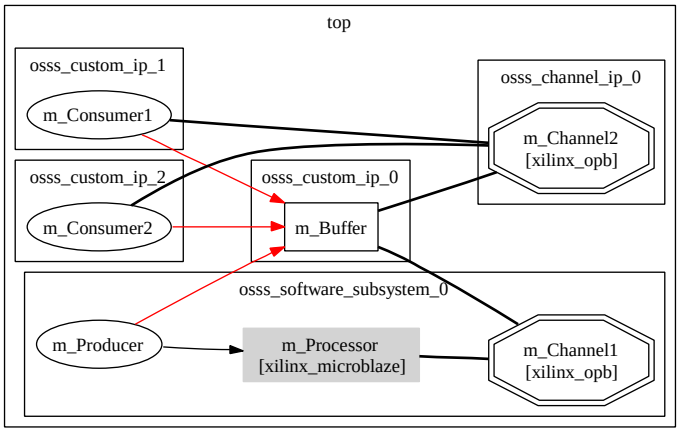
connected to an OPB.

Out of the Xilinx specific components the MHS (**M**icroprocessor **H**ardware **S**pecification) and the MSS (**M**icroprocessor **S**oftware **S**pecification) file for the EDK is generated. The Xilinx ML401 system description is used to generate a target platform dependent UCF (**U**ser **C**onstraint **F**ile) that contains the physical position of the clock, reset and external memory pins. In order to pack the user-defined IP for using it in the Xilinx EDK an MPD (**M**icroprocessor **P**eripheral **D**efinition) and a PAO (**P**eripheral **A**nalyze **O**rder) file are generated.

The OSSS ACI (**A**rchitectural **C**ontext **I**nformation) file is basically generated from the information gained during architectural context extraction and gets enriched by some of the target platform dependent information. In general it contains all the information about the internal structure of the user-defined hardware part of the system that is further processed by the high level synthesis tool FOSSY. Additionally it contains information about how this user-defined hardware is integrated and connected to the overall system architecture defined by the MHS.




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




(b) Partitioning view

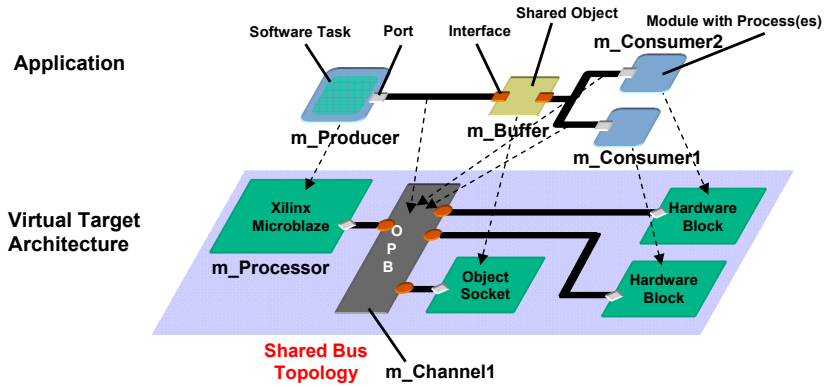
Figure 8.4: Partitioning of the producer/consumer example mapping alternative C

In the following sections the generated data will be described in more detail.

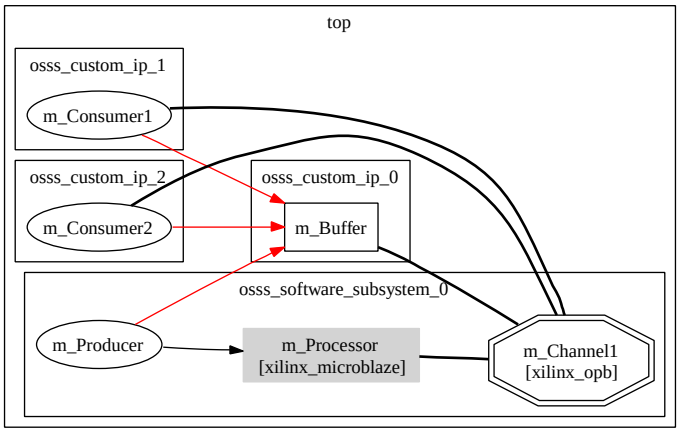
### 8.1.2 MHS and MSS Generation

The Xilinx EDK builds on top of the Xilinx ISE and can be regarded as an entry for the ISE synthesis flow. The aim of the EDK is the integration of various IP components including embedded processors, DSP blocks, peripherals and communication IPs. The EDK itself includes a library of several IP components including the MicroBlaze embedded soft core processor plus several peripherals which can be interconnected by using the IBM CoreConnect technology. Since the MicroBlaze is a soft processor core it can be fully customized by the designer instantiating it. By using the Xilinx EDK a designer can assemble an architecture consisting out of different IP components including user-defined hardware blocks. After the assembly and configuration of the desired architecture the Xilinx ISE is used to synthesise and download the whole design to an FPGA.

In the ICODES project the Xilinx EDK serves three purposes:




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




(b) Partitioning view

Figure 8.5: Partitioning of the producer/consumer example mapping alternative B

1. It is used to generate the hardware architecture of the hardware/software design. This includes the generation of black-box components for the integration of the FOSSY synthesis output into the overall system architecture.

2. Provides the cross-compiler tool chain for the MicroBlaze

3. Serves as a front-end to the Xilinx or Synplicity synthesis tools

During this section we will concentrate on the creation of an EDK project by means of generating an MHS and an MSS file.

The MHS file defines the system architecture consisting out of peripherals, and embedded processors. It also defines the connectivity of the system, the address map of each peripheral in the system, and the configurable options for each peripheral. It is also possible to specify multiple processor instances connected to one or more peripherals through one or more buses and bridges. Since the MHS file is a simple text file it can be created and examined easily by using a text editor.

This simple file format enables the automatic creation of an MHS. The generation of an MHS file is performed by the building blocks of the Virtual Target Architecture. Each class of the architecture class library has a method that appends its configuration to the MHS file. The MHS file as well as the architectural context information is generated during the end of elaboration phase.

The Xilinx EDK Platform Generator tool (Platgen) creates the hardware platform using the MHS file as input. Platgen creates netlist files in various formats such as EDIF and Xilinx proprietary NGC, support files for downstream tools, and top level HDL wrappers to allow the addition of user-defined components to the automatically generated hardware platform.

The software platform is defined by the Microprocessor Software Specification (MSS) file. The MSS file defines driver and library customization parameters for peripherals, processor customization parameters, standard input/output devices, interrupt handler routines, and other related software features. The MSS file is also a simple text file and thus can be treated like the MHS.

The creation of the MSS file is analogue to the MHS file generation. It is performed during the end of elaboration phase of the SystemC kernel. Each class of the architecture class library has a method that appends information about its software driver to the MSS file.

The MHS and the MSS file are inputs to the Xilinx EDK Library Generator tool (Libgen) for customization of drivers, libraries, and interrupt-handlers.

### 8.1.3 UCF Generation

Details such as I/O pin mappings and timing constraints can not be expressed in Verilog or VHDL, but are nonetheless important considerations when implementing the design on real hardware (i.e. an FPGA). The UCF file is an ASCII file specifying physical constraints on the logical design.

In the proposed synthesis flow the UCF file is generated from the information contained in the top-level entity of the OSSS design on the *Architecture Layer*. Its main purpose is the mapping of logical ports defined in the top-level design (such as clock, reset or user-defined I/O pins) to the physical port locations of the FPGA. Most of this mapping can be done automatically when the designer specifies the kind of prototyping board he uses.

The UCF file is generated during the end of elaboration phase of the SystemC kernel. It is used as input to the Xilinx ISE Synthesis tools (more precisely the NGDBuild tool that is used to build up the internal representation for the mapping and the place & route phase).

### 8.1.4 MPD and PAO Generation

The MPD and the PAO files are both needed to wrap the VHDL files generated by FOSSY. They are used to integrate user-defined hardware blocks as Xilinx EDK compatible IP cores and to define the further processing of the VHDL files by the backend RTL synthesis tools.

The Microprocessor Peripheral Definition (MPD) file defines the interface of the peripheral. An MPD file has the following characteristics:

- Lists ports and default connectivity for bus interfaces
- Lists parameters and default values
- Any MPD parameter is overwritten by the equivalent MHS assignment

Thus the MPD file defines the MHS representation of the user-defined hardware generated by FOSSY.

A Peripheral Analyze Order (PAO) file contains a list of HDL files that are needed for synthesis, and defines the analysis order for compilation. This information is needed by the backend RTL synthesis tools that are used to process the FOSSY generated VHDL code. Additionally it defines the compilation order of the software driver during cross-compilation for the target CPU.

### 8.1.5 OSSS ACI Generation

In order to perform the hardware/software interface synthesis the following information needs to be provided by the architectural context extraction:

- The design structure on the *Application Layer* of the OSSS design at the end of elaboration phase. It consists out of `osss_software_task`s and `sc_module`s containing `osss_port`s bound to Shared Objects.
- The design structure on the OSSS *Virtual Target Architecture Layer*. It consists of different kinds of architecture building blocks and connections to either OSSS-Channels or signals. Together with the design structure on the *Application Layer* model the mapping is retained. This includes the channel bindings, i.e. the mapping of the communication links to OSSS-Channels. This implies the number of clients to a Shared Object.
- Object IDs for each Shared Object plugged into an `osss_object_socket`
- Method IDs for each method of a Shared Object
- Client IDs for each client process performing calls on a Shared Object

- OSSS-Channel transactor specific parameters: E.g. channel bit width of a point-to-point channel and each client's address range for bus transactors. The address range of each slave interface of an `osss_object_socket` is calculated in subject to the number of clients hidden by the corresponding bus.

This Architectural Context Information (ACI) serves as input for the high-level synthesis tool FOSSY (see Section 8.3) and the RMI-Types generator for the customization of the OSSS software library (see Section 8.2).

## 8.2 Software Library Synthesis

The synthesis flow for the hardware and the software part of an OSSS design are separated from each other. The user-defined hardware of the design is processed by FOSSY (see Section 8.3) in conjunction with the Xilinx EDK & ISE as described above. The user-defined software of the design needs to be cross-compiled for the processors defined by the hardware platform. Up to now only the MicroBlaze soft processor core is supported.

For the cross-compilation the MicroBlaze gcc (`mb-gcc`) that uses a `gcc` 3.4.1 front-end is used. The Xilinx EDK provides the complete GNU tool chain for the MicroBlaze processor. The input for the MicroBlaze cross-compiler is the entire code that is used inside an `osss_software_task`. When compiling an `osss_software_task` for the MicroBlaze the *OSSS Software Library* needs to be included. It contains the OSSS software API and the OSSS RMI protocol stack. For a more detailed description of the OSSS software library refer to Section 8.2.

As shown in Figure 8.1 the *OSSS Software Library Generator* is used to add design specific code to the *OSSS Software Library*. This design specific code consists out of unique IDs (client ID, object ID, method ID). This information is necessary to set up a working hardware/software communication. Both the OSSS RMI protocol stack running on the MicroBlaze and the implementation of the hardware interface performed by FOSSY need to share this specific information.

After the `osss_software_task` (including the OSSS Software Library) has been compiled by the `mb-gcc`, it is linked by the Xilinx library. The Xilinx library is generated and compiled by the Xilinx EDK Library Generator by using the `mb-gcc` as well. It includes code for performing I/O with the MicroBlaze processor and communication with Xilinx specific peripherals (e.g. a `printf(...)` function is provided for sending strings to the UART).

The linking is controlled by a Linker Script that is needed because the MicroBlaze runs in "stand-alone" mode, i.e. without an operating system. The Linker Script defines the memory layout of the application. It defines where the different sections for services like memory allocation and object destruction are mapped in the memory. Additionally it defines the stack and the heap size of the application. Linker Scripts for program execution from internal (Block RAM only) and/or external (on board DDRAM) memories are provided.

After linking has been performed an executable file in the Executable and Link Format (ELF) is generated. When running an application from the internal Block RAM the Xilinx Bitstream Initialiser is used to initialize the Block RAMs with the context of the ELF file. After initializing the `system.BIT` with ELF data from the software compilation the result is a bitstream with initialized Block RAM resources, called `download.BIT`. This configuration file can be downloaded to the FPGA via JTAG by using the Xilinx iMPACT tool. When the resulting application is too big to be executed from the block RAM only, it needs to be loaded into the external RAM. This can be performed by a special boot loader via the RS232 port or by the Xilinx Microprocessor Debugger (`XMD`, with `XDM_stub`) in conjunction with the GNU Debugger (GDB) via JTAG.

### 8.2.1 Supported Software Language Subset

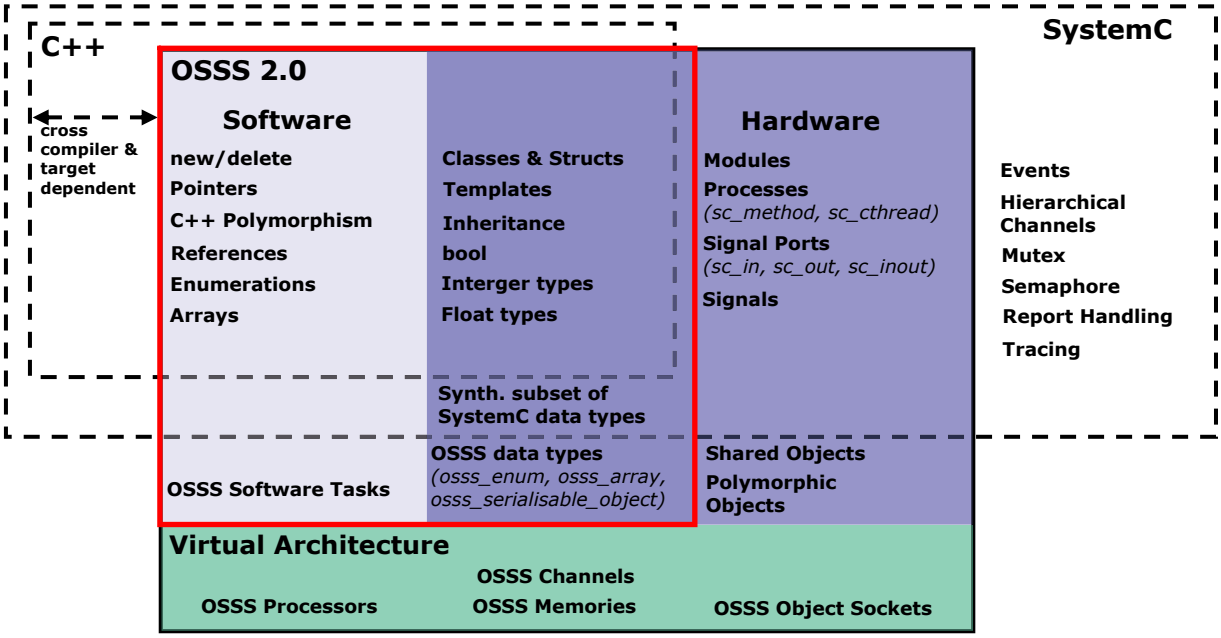



Figure 8.6: The OSSS 2.2.0 software language subsets

Figure 8.6 shows the relation of OSSS to C++ and SystemC. OSSS is divided into three different subsets: a software description subset, a hardware description subset and an architecture description subset. Each of these subsets defines which modelling elements or language features are allowed in each domain (software, hardware or architecture). There is one intersection between the software and the hardware domain. This intersection includes the language features that may be used in both the hardware and the software domain. This intersection is very important because it defines the hardware/software interface modelling elements. For instance only the data types situated in the hardware/software intersection may be used for hardware/software communication.

The `osss_software_task` is the foundation for the software part of each OSSS design. The red square in Figure 8.6 shows the definition of the language subset that is allowed

inside `osss_software_tasks`.

Each software task has a single `main()` method which is the root function or entry point of each task. Therefore the `main()` method contains the software only part. In principal the whole ISO C++ [cpp98] language can be used inside of it. Regrettably this is only the best-case expectation because the capabilities of different software compilers and the architectural differences between the host processor (used for the simulation of the hardware/software design) and the target processor have also to be considered. This is important, because the functional behaviour of the software code running on the host or the target processor should be the same.

Therefore, we assume that the compiler front-end used to compile code for the host machine is the same as used for the cross-compilation. The only software compiler front-end supported by OSSS is the GNU Compiler Collection for C/C++ (`gcc` respectively `g++`) [gccb]. We currently use a `gcc` 3.4.4 or higher for the compilation on the host machine. The cross-compiler for the MicroBlaze processor is the `mb-gcc` that is based on a `gcc` 3.4.1 [mbg]. By using the same compiler front-end for the simulation on the host and the execution on the target processor we allow the C++ language subset supported by the `gcc`.

Furthermore, another issue concerning the functional equivalent behaviour is the usage of non-portable language features of C++. In the software subset of OSSS some of the non-portable language features of C++ are allowed, but the OSSS design methodology does not guide the software designer how to port his code. In consequence the designer has the responsibility to port the software part from the host processor to the target processor "by hand".

A precise definition of the OSSS language and its software subset will be provided with an *OSSS Language Reference Manual* at a later date.

### 8.2.2 The Software Interface Synthesis for HW/SW Communication

The software synthesis technology for hardware/software interfaces is based on the Remote Method Invocation (RMI) concept. For detailed information concerning the RMI concept please refer to Section 6.2.4.

Figure 8.7 shows an `osss_software_task` that has been mapped onto a `xilinx_microblaze` soft processor core. The `osss_software_task` provides a `main()` method which is the entry point for the user-defined software. It makes use of an `osss_port<osss_rmi_if<IF> >` for calling a method on a specific interface (`IF`) which is implemented by user-defined hardware (not shown in Figure 8.7). As communication medium it uses a `xilinx_opb_channel` that is wrapped by an `osss_rmi_channel<...>` to make it RMI capable.

After the architectural context extraction phase of the synthesis design flow (as described in Section 8.1) a Xilinx MicroBlaze soft processor core connected to an On-Chip Peripheral Bus (OPB) is generated. Figure 8.8 shows the synthesised architecture corresponding to the design shown in Figure 8.7. The `xilinx_microblaze` architecture building block contains its own local memory, a hardware timer, access to a shared UART and shared

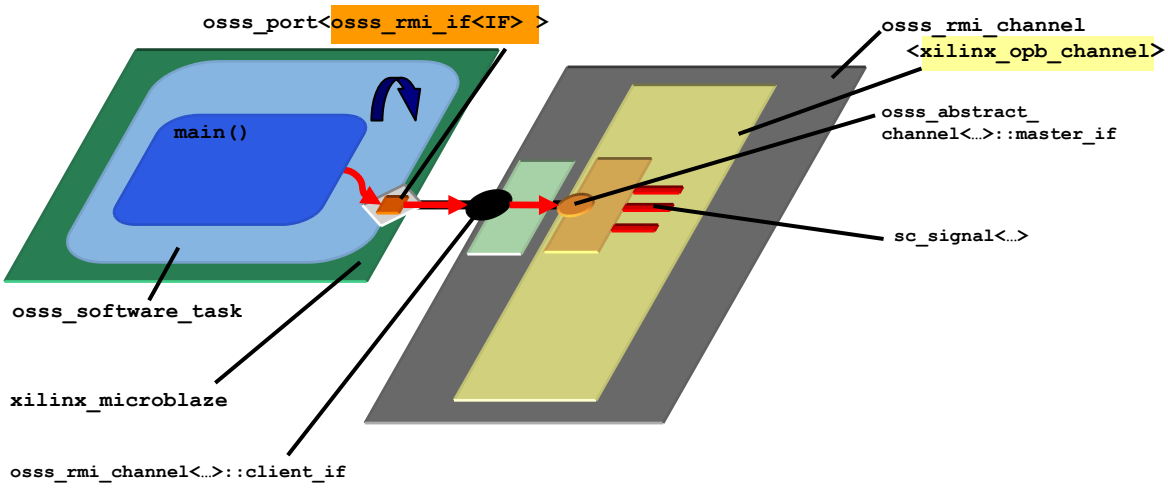



Figure 8.7: The considered software interface part for hardware/software communication

debug facilities.

The `xilinx_opb_channel` in Figure 8.8 shows two different versions. The OPB1 is connected to the native DOPB and IOPB port of the MicroBlaze, while OPB2 is connected to a FSL2OPB bridge. The difference between these two buses comes from the ability to initiate burst transfers through the FSL2OPB Bridge.

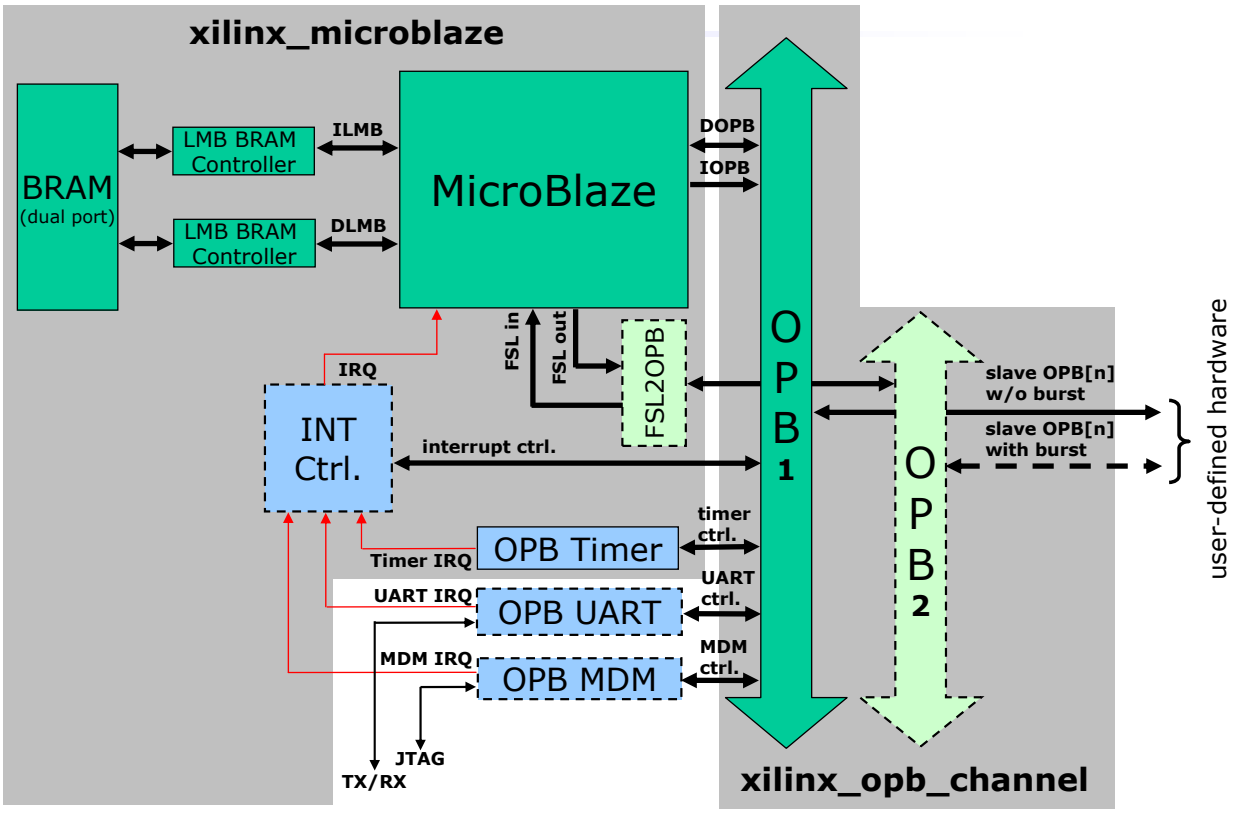



Figure 8.8: Synthesised architecture corresponding to Figure 8.7

### 8.2.3 The OSSS Software Library

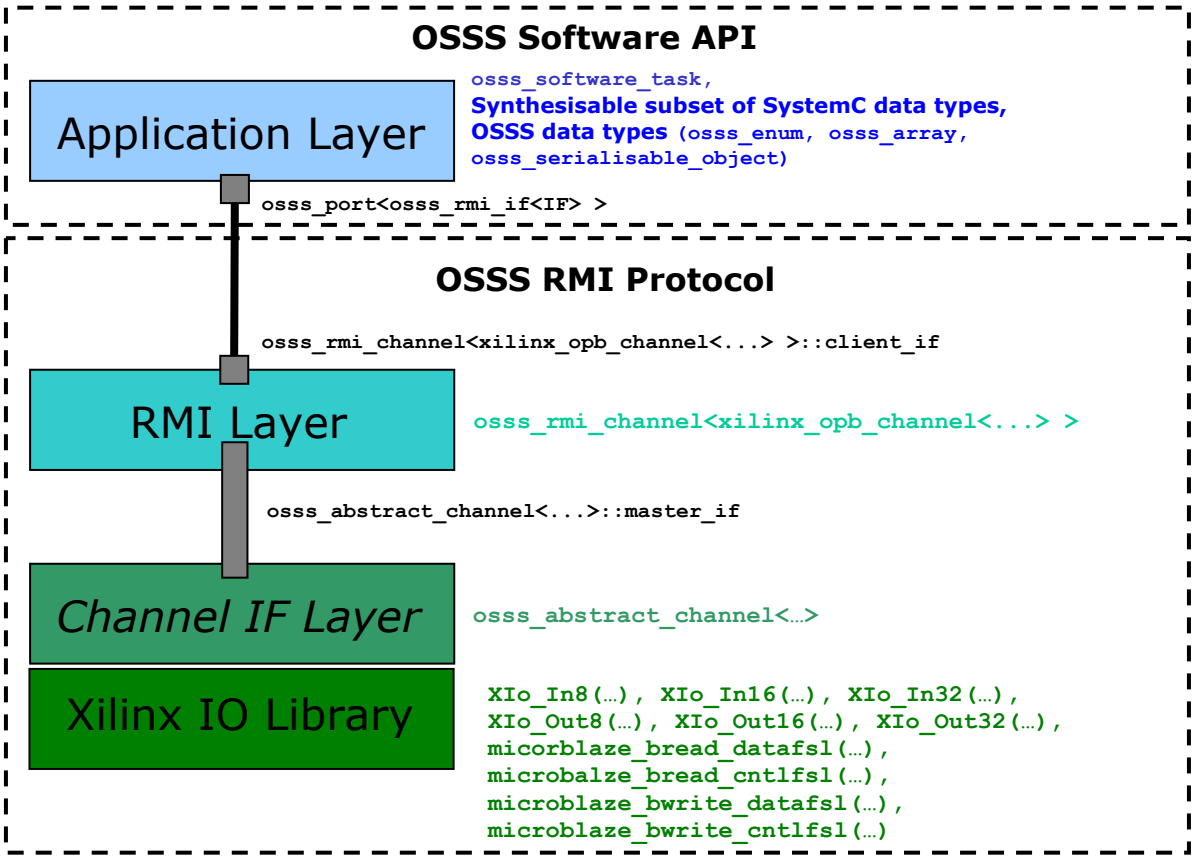



Figure 8.9: The OSSS software library used on the Xilinx MicroBlaze processor

The OSSS software library provides features and functionalities for the software part of an OSSS design that are essential when running it on the target processor. It is divided into the Software API and the RMI protocol stack, see Figure 8.9:

1. The OSSS Software API (software related parts of the *Application Layer*) provides:
   - The same interface to the software designer during simulation and synthesis (i.e. the application designer does not need to change any code of the software part of the system concerning:
     (a) the interaction with the `osss_software_task`,
     (b) the communication with the hardware through `osss_port<...>`
   - Software execution environment (`osss_software_task`)
   - Data types for hardware/software communication (Synthesisable subset of SystemC data types and OSSS data types)
   - Interface to the OSSS RMI protocol stack (`osss_port<osss_rmi_if<...> >`)
2. OSSS RMI protocol stack performing the hardware/software communication protocol

The OSSS Software API has been designed to be as target processor independent as possible. Even most parts of the underlying OSSS RMI protocol stack are target processor independent. Only the bottommost I/O layer of the OSSS software library is target processor dependent.

After Architectural Context Extraction the *OSSS Software Library Generator* adds necessary target specific information to this software library.

Afterwards, the OSSS software library is linked to the user code when the cross-compilation for the specific target processor (i.e. the Xilinx MicroBlaze processor) is performed.

The following subsections explain the different layers of the OSSS software library.

### Application Layer

The *Application Layer* of the *OSSS Software Library* contains the `osss_software_task` that is the execution environment for all software parts of the design. Up to now only a single `osss_software_task` can be mapped on a processor. Therefore the `osss_software_task` is basically a wrapper for the native `int main()` entry for a C/C++ program running on a processor.

The software timing during simulation as defined by EET (Estimated Execution Time) blocks are eliminated in the OSSS software library. When running the cross-compiled code on the target CPU itself the timing behaviour is inherently defined by the execution order and the interpretation of the instructions on the target CPU.

Besides the software task the *Application Layer* contains the synthesisable subset of SystemC data types and some OSSS data types. Basically all data types of OSSS hardware/software intersection (see Figure 8.6) are allowed for the hardware/software communication. The OSSS data types or container `osss_enum<...>` and `osss_array<...>` are serialisable replacements for the C++ built-in `enum` type and the general array types.

The `osss_serialisable_object` is the base class for all user-defined data types that need to be used for hardware/software communication. User-defined data type that should be used for the hardware/software communication must be derived from the `osss_serialisable_object` class.

When transferring data between hardware and software it must be ensured that the interpretation of the type is the same. Therefore the byte ordering (big- or little-endian) is handled by the serialisable object with respect to the byte ordering supported by the processor. The byte ordering in the `osss_serialisable_object` is always big-endian. The necessary conversions when the software is executed on a little-endian processor are done automatically by the serialisable object.

For the communication of a software task with objects implemented in hardware the

`osss_port<osss_rmi_if<IF> >` is also part of the *Application Layer*. The OSSS port defines the interface to the RMI Layer.

### RMI Layer

Since a software task can only be the initiator of a communication only the initiator part of the RMI is implemented in the OSSS RMI protocol of the OSSS software library.

The `osss_port<osss_rmi_if<IF> >` shown in Figure 8.9 represents the interface between the OSSS Software API and the OSSS RMI protocol stack.

The designer has to supply the `osss_rmi_if<...>` container for each interface implemented by the user-defined class inside a Shared Object.

This interface has to contain stubs for all methods of the type of the interface class. The stubs are generated by the `OSSS_METHOD_STUB(...)` and the `OSSS_METHOD_VOID_STUB(...)` macros. The OSSS RMI layer of the OSSS software library provides the same macros in order to reuse the code of all user-defined `osss_rmi_if<...>` containers.

The communication of the MicroBlaze processor with user-defined hardware blocks is performed through the OPB. Therefore the RMI layer of the OSSS software library provides the `osss_rmi_channel<xilinx_opb_channel<...> >::client_if` interface only. As a consequence the *Channel Layer* interface needs to provide the `osss_abstract_channel<...>::master_if` interface only.

In general the implementation of the RMI layer of the OSSS software library is analogue to the implementation of this layer in the OSSS simulation library. This layer of the OSSS software library is implemented with the intention to be as portable as possible between at least all cross-compilers using a gcc compiler front-end.

### Channel Layer

The *Channel Layer* of the OSSS software library shown in Figure 8.9 is the most processor specific layer of the RMI protocol stack. The *Channel Interface Layer* is used to encapsulate the processor dependent implementation from the portable (or processor independent) RMI layer.

The `master_if` implemented in the *Channel Layer* performs the translation from the abstract interface using `read_blocking(...)` and `write_blocking(...)` methods to the Xilinx MicroBlaze specific input/output mechanisms.

The chosen output mechanism of the MicroBlaze depends on whether burst transfers are used or not used. Xilinx MicroBlaze specific basic input/output macros are defined that can be used to access slave components which are connected to the OPB and are properly mapped in its memory space.

## 8.3 High-Level Synthesis

The high-level synthesis is performed by FOSSY (**F**unctional **O**ldenburg **S**ystem **SY**nthesiser). The hardware interface synthesis affects two entities of a design, namely Shared Objects and hardware clients of Shared Objects. The mapping of the *Application Layer* to the *Virtual Architecture Layer* guides the hardware interface synthesis process by means of well defined communication refinement rules.

### 8.3.1 FOSSY

Figure 8.10 gives an overview of the FOSSY tool chain and provides a more detailed view than shown in the overall synthesis flow in Figure 8.1.

FOSSY is written in the pure functional programming language Haskell [has].

Since both the internal FOSSY data structure and the intermediate format of the front end are based on the same ISO/IEC C++ standard [cpp98] the implementation of the adaptation layer is straightforward. The EDG front end parser (called cc2cil) [edg] is used and augmented with a thin layer converting the front end specific intermediate format to XML (called cil2xml). Large parts of the logic of this conversion layer are automatically generated from the Haskell type structure.

Haskell values conforming to the type system representing the C++/SystemC/OSSS grammar are written to and read from XML files. The DTD describing the structure of the XML is derived from the type system automatically. The human readable XML data is the input for FOSSY and is open to a range of existing XML tools.

In order to simplify and to accelerate the high-level synthesis a special OSSS synthesis library is used as FOSSY input. This synthesis library includes reduced header files for all OSSS modelling elements. The same reduced header files are provided for the SystemC library. Since FOSSY does not rely on the analysis of all method bodies most of them can be omitted. This results in a significant speed up during the parsing step.

Besides the OSSS design on *Application Layer* information from the *Virtual Target Architecture Layer* are needed by FOSSY. As described in Section 8.1 the architectural context extraction step is used to generate architectural context information (ACI) that are stored and forwarded to FOSSY in an XML format.

The ACI file contains the following information:

1. The channel bindings, i.e. the mapping of the communication links to OSSS-Channels. This implies the number of clients to a Shared Object.
2. Transactor specific parameters e.g. channel bit width of a point-to-point channel and each client's address range for bus transactors. Note that the address ranges imply the Shared Object ID and the client IDs, since each client has a unique address range for each of its serving Shared Objects

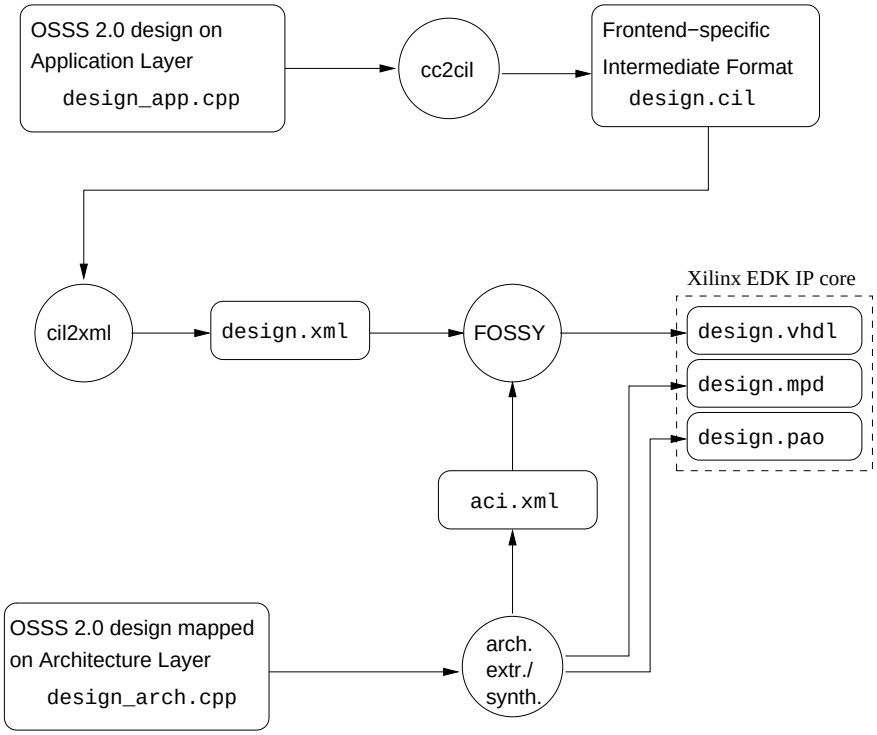



Figure 8.10: FOSSY high-level synthesiser tool chain

3. Method IDs for each method of a Shared Object

4. The layout of the transmitted, i.e. serialised data

The output of FOSSY is a synthesisable VHDL description of the user-defined hardware part of the OSSS design. For using this output in the Xilinx backend synthesis flow it needs to be "imported" to the Xilinx EDK project. As shown in Figure 8.1 the FOSSY synthesis output is stored in the user repository of the EDK project and thus constitutes a part of the overall OSSS design architecture. From the Xilinx EDK's point of view the output generated by FOSSY is treated like a 3rd party IP component.

### 8.3.2 Hardware Interface Synthesis

Method-based communication between SW and HW or HW and HW is uniformly done via the RMI protocol. The callee of such a communication must always be a Shared Object. Therefore, the hardware interface synthesis regarding the Shared Object is independent of whether the caller is a HW module or a SW task.

Figure 8.11 shows the domain of the hardware interface synthesis for hardware/software communication. A Shared Object which is wrapped by an `osss_object_socket` is connected to a `xilinx_opb_channel` on the left and an `osss_simple_point_to_point_channel` on the right side. Therefore the Shared Object (or more precisely the `UserClass` inside the Shared Object) can be called by any master connected to the OPB (this includes software running on a MicroBlaze) and by the hardware client connected to the point-to-point channel. The `osss_object_socket_port` along with the transactor of the `osss_rmi_channel` and the

transactor of the `xilinx_opb_channel` or the `osss_simple_point_to_point_channel` describes the interface of the Shared Object. For a more detailed description of the elements shown in Figure 8.11 refer to Section 6.2.4.

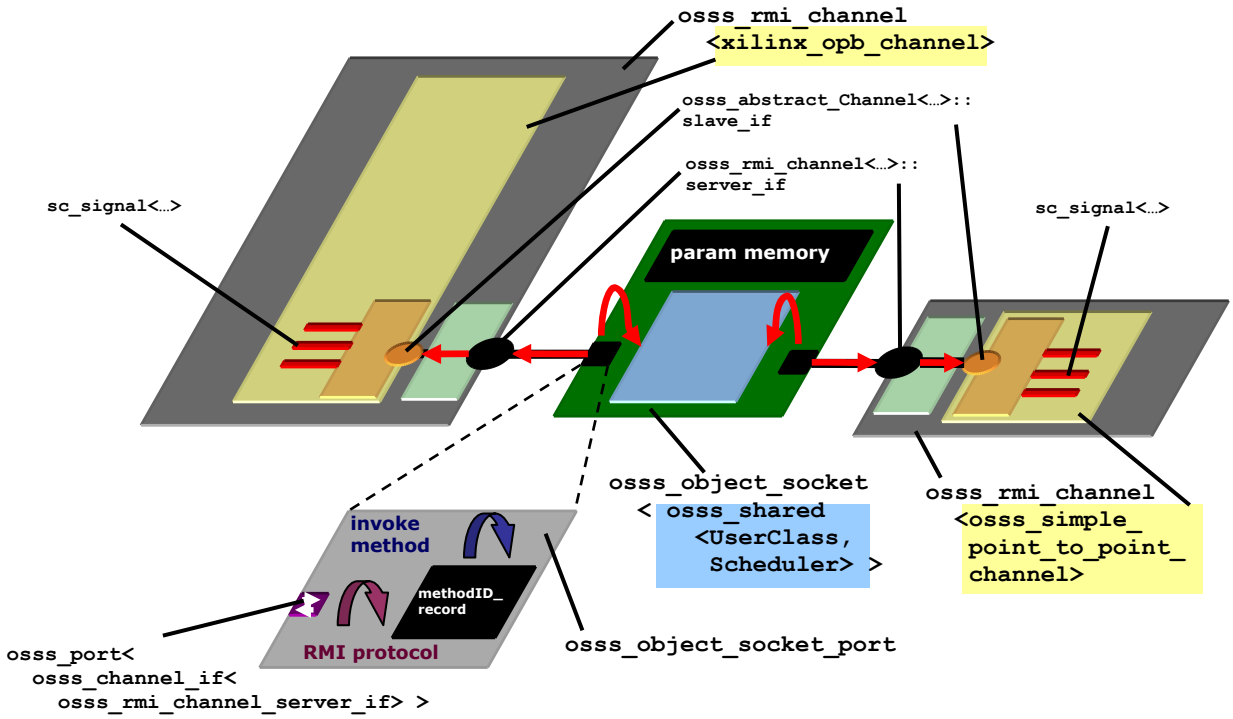



Figure 8.11: The considered hardware interface part for HW/SW and HW/HW communication

The requirements of the Shared Objects interface synthesis strategy are:

- Connection to existing busses (for the communication with the CPU/SW)
- Handling the RMI protocol

The extended, i.e. hardware/software-enabled, synthesised Shared Object from Figure 8.11 is shown in Figure 8.12 (in dark blue displayed the new parts of the Shared Object implementation).
In the figure a bus is connected to the interface `IF1` and a hardware client is directly connected to `IF2`. In other words, the interfaces `IF1` and `IF2` are the connections of the Shared Object to its clients. These interfaces are mainly necessary due to the first requirement (connection to existing buses). The controller within the Shared Object handles the RMI protocol and manages concurrent requests with the help of the scheduler. Hence it is the controller and the interfaces which form the actual hardware interface.

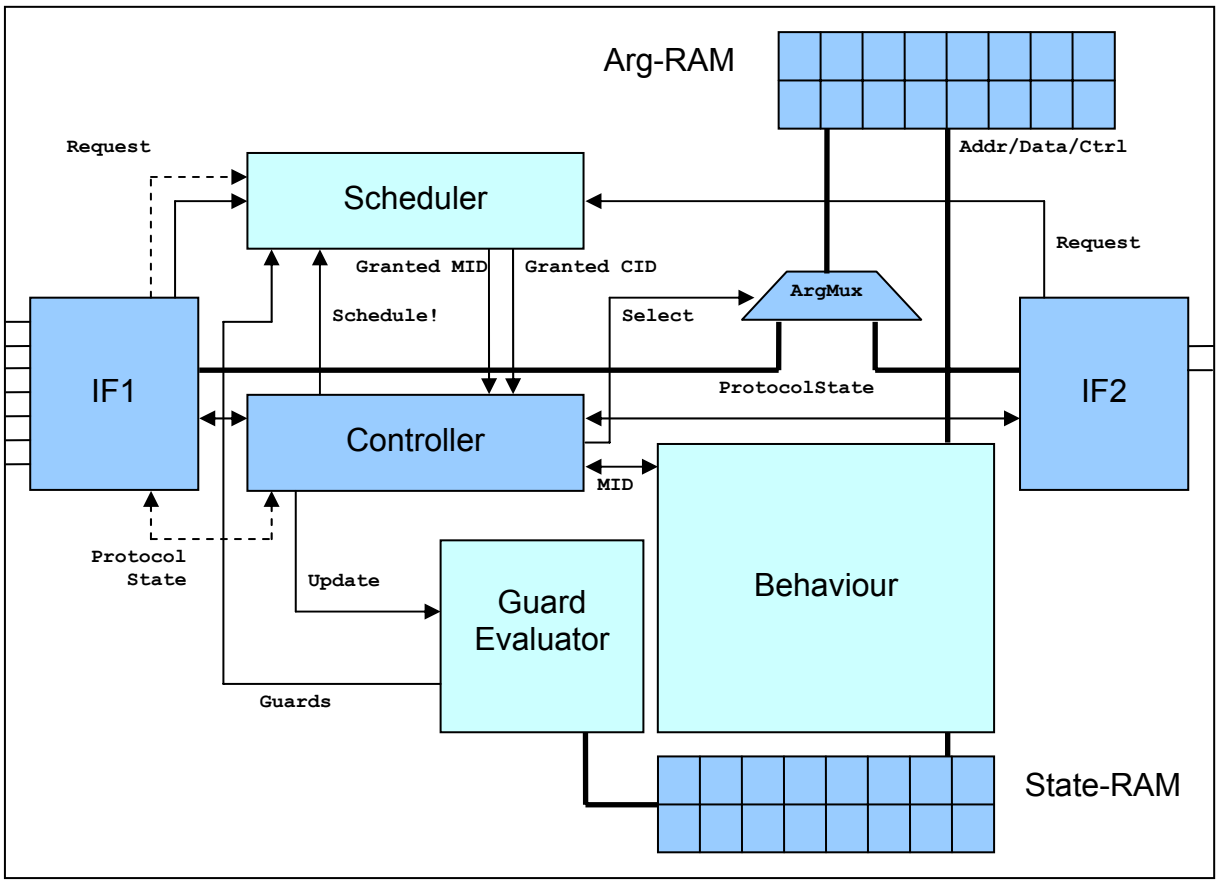



Figure 8.12: Structure of a synthesised Shared Object from Figure 8.11

The generated hardware structure only depends on the total number of clients, but neither on their types, i.e. hardware or software nor on their specific connection to the Shared Object.

The inner structure of a certain interface block, e.g. for an interface block which connects to an OPB, is relatively fixed. It only depends on a few parameters like the number of clients and their corresponding addresses. The same is true for point to point channels where the only parameter is the bit width. Therefore, a prototypical synthesis approach is to instantiate pre-designed and properly parameterised hardware structures.

For the synthesis of the hardware client interface a predefined macro that represents combination of the RMI- and the master transactor is used.

For the first synthesis tool prototype, only certain predefined OSSS-Channels over which RMI can be performed, are allowed. Hence, the synthesis tool needs to know on what kind of predefined channel a communication link is mapped, but it does not process the channel-internal implementation.

## 8.4 Synthesis Subset

In general an OSSS application layer design is synthesisable[1]. This section gives a short overview of the "really" synthesisable SystemC/OSSS language constructs accepted by the current implementation of the synthesis tool which will be referred to as *Fossy* or simply "the synthesiser".

### 8.4.1 Compatibility to the SystemC Synthesisable Subset

*Fossy* is quite compatible to the SystemC Synthesisable Subset. Table 8.1 lists all important features of the synthesis subset. Unsupported features or features with restricted synthesis semantics are commented.

| **C++/SystemC™ feature** | | | **Comment** |
|---|---|---|---|
| Translation units | | √ | Only one translation unit allowed. |
| Modules | | | |
| | Definition: `SC_MODULE`, `sc_module` inheritance | √ | |
| | Members: signal, sub-module, ctors, `HAS_PROCESS` | √ | |
| | Ports/Signals: `sc_signal`, `sc_in`, `sc_out`, `sc_inout`, `sc_in_clk` | √ | |
| | Ports/Signals: `sc_out_clk`, `sc_inout_clk` | – | Not supported, but deprecated anyway. |
| | Ports/Signals: `*_resolved`, `*_rv` | – | Not supported. |
| | Ctor: w/ and w/o `SC_CTOR` macro | √ | |
| | Deriving | √ | |
| Datatypes | | | |
| | Integral types | √ | |
| | Integral promotion, arith. conversion | √ | |
| | Operators | √ | |
| | Compounds: arrays, enums, class/struct/unions, functions | √ | No pointers and no references supported. |
| | SysC: `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint` | √ | |
| | SysC: fixed-point types | – | Not supported. |
| | SysC: `sc_bv` | √ | |
| | SysC: `sc_logic` | √ | Converted to `sc_bv<1>` internally. |
| | SysC: `sc_lv` | – | Not supported. |
| | SysC: arithmetic operators | √ | |
| | SysC: bitwise operators | √ | |
| | SysC: relational oparators | √ | |
| | SysC: shift operators | √ | |
| | SysC: assignment operators | √ | No chained assignments, LHS must be side effect-free. |

[1] This should be no surprise since OSSS by definition is a synthesis subset.

| | | | |
|---|---|---|---|
| | SysC: bit select operators | √ | |
| | SysC: part select operators | √ | Reverse ranges not supported. |
| | SysC: concatenation operators | √ | Assignment to concats not supported. |
| | SysC: conversion to C integral (`to_int()`, `to_bool()` etc.) | √ | |
| | SysC: additional methods: `iszero()`, `sign()`, `bit()`, `reverse()`, etc. | – | accepted, but not fully implemented |
| Declarations | | | |
| | typedef | √ | |
| | enums | √ | |
| | aggregates | – | Not supported. Reduced support for ROM initialization needed. |
| | arrays | √ | |
| | references | – | Copy-in copy-out support for functions/methods. |
| | pointers | – | Pointers allowed for sub-module instantiation. |
| Expressions | | √ | `new`/`delete`/pointers not supported, no chained assignments. `new` allowed for sub-module instantiation. |
| Functions | | √ | |
| Statements | | √ | |
| Processes | | √ | No `SC_THREAD` supported. |
| Sub-module instantiation | | √ | No support for positional port binding. |
| Namespaces | | √ | |
| Classes | | | |
| | Member functions | √ | No wait supported. |
| | Member vars | √ | |
| | Inheritance | √ | |
| | Abstract classes | √ | |
| | Constructors | √ | Restrictions with default arguments (no default arguments allowed). |
| Overloading | | √ | |
| Templates | | √ | |
| Pre-processing directives | | √ | |

Table 8.1: *Fossy* SystemC Synthesisable Subset support

### 8.4.2 Source Code Organisation

An OSSS design should be divided into several header (`.h`) and source files (`.cc`). Usually each header/source file pair should contain one module declaration/definition. An exception to this rule are templates. Templates should be completely described in the header file. A corresponding source file is not necessary (more precisely: not possible) in this case.

*Limitation:* Currently *Fossy* can only process a single translation unit, i.e. a single `.cc` file. As a workaround you can write a special main file, e.g. `fossy_main.cc` which includes all necessary `.cc` files. You should *not* `#include` any `.cc` files in header files.

An example of such a structure is shown in Listing 8.2 and Listing 8.3. The synthesis always needs a top-level **module** to start from, i.e. in order to do something useful with the synthesiser, the design must contain at least one module.

```
1  #include <systemc.h>
2  #include <osss.h>
3
4  SC_MODULE(SomeModule)
5  {
6    sc_in<bool>    somePort;
7
8    void someProcess();
9
10   SC_CTOR(SomeModule)
11   {
12     SC_METHOD(someProcess);
13     sensitive << somePort;
14   }
15 };
```

Listing 8.2: General structure of the *Fossy* input, `SomeModule.h`

```
1  #include "SomeModule.h"
2
3  void
4  SomeModule::someProcess()
5  {
6    /* do something useful here */
7  }
```

Listing 8.3: Source file `SomeModule.cc` corresponding to Listing 8.2

### 8.4.3 Design Hierarchy

#### Modules

`SC_MODULE` is supported. An `SC_MODULE` can be defined via the macro `SC_MODULE` or by manually deriving from `sc_module`. If the latter form is used, the macro `SC_HAS_PROCESS(<ModuleName>)` has to inserted. An example is shown in Listing 8.4.

```
1  #include <systemc.h>
2
3  SC_MODULE(myModule1)
4  {
5    // ...
6  };
7
```

```
8  struct myModule2 : public sc_module
9  {
10   // ...
11
12   // Required if this module contains processes
13   SC_HAS_PROCESS(myModule2);
14 };
```

Listing 8.4: A synthesisable module definition

*Limitation:* Inheritance is not supported for user modules.

*Limitation:* Every non-process non-void method of a module, i.e. a method which is not a process and which does return a value, must have exactly one `return` statement as its last statement. The returned expression must be a variable.

*Note:* Each module must have exactly one constructor.

### Constructors

A module constructor can be defined via the macro `SC_CTOR` or it can be defined manually. An example is given in Listing 8.5 and Listing 8.6

```
1  // Begin of myModule1.h
2  #include <systemc.h>
3
4  SC_MODULE(myModule1)
5  {
6    sc_in<bool>    myPort;
7
8    // Alternative 1:
9    // Use SC_CTOR and place the body in the
10   // header file
11   SC_CTOR(myModule1)
12   : myPort("myPort")
13   , // ... more initialisers
14   {
15     /* constructor body */
16   }
17
18   // Alternative 2
19   // Manually specify the constructor and
20   // place the body in the header file
21   myModule(sc_module_name name)
22   : myPort("myPort")
23   , // ... more initialisers
24   {
25     /* constructor body */
26   }
27
28   // Alternative 3
29   // Manually specify the constructor and
30   // place the body in the source file
31   myModule(sc_module_name);
```

```
32    // the body is located in the .cc file
33
34    // Alternative 4:
35    // Use SC_CTOR and place the body in the
36    // source file
37    SC_CTOR(myModule1);
38    // the body is located in the .cc file
39    // (and looks exactly like the one for
40    // Alternative 3).
41
42  };
43  // End of myModule1.h
```

Listing 8.5: Synthesisable module constructors, header file

```
1   // Begin of myModule1.cc:
2   #include "myModule1.h"
3
4     myModule1::myModule1(sc_module_name)
5     : myPort("myPort")
6     , // ... more initialisers
7     {
8       /* constructor body */
9     }
10  // End of myModule1.cc
```

Listing 8.6: Synthesisable module constructors, source file

*Limitation:* Module constructors must have exactly one parameter: the module name, which must be of type `sc_module_name`.

*Limitation:* Module constructors must not contain any control structures (loops, if-then-else) or any blocks.

*Note:* The name which is supplied as constructor argument to named objects like modules, ports, signals is ignored by *Fossy*. The resulting name will always be the attribute name.

### Ports

The following ports are allowed:

- `sc_in<T>`
- `sc_out<T>`
- `sc_inout<T>`
- `osss_port_to_shared<IF>`

Ports may be used directly or by pointer. If the pointer variant is used, the port may be initialised in the intialiser list or in the constructor body as shown in Listing 8.7.

```
#include <systemc.h>

SC_MODULE(Sub)
{
  sc_in<bool>  port1,    // used directly
               *port2,   // by pointer
               *port3;   // by pointer

  SC_CTOR(Sub)
  : port1("port1")
  , port2(new sc_in<bool>("port2"))    // initialised in the
                                       // initialiser list
  {
    port3 = new sc_in<bool>("port3"); // assigned in the body
  }
};
```

Listing 8.7: Synthesisable ports

The type `T` may be any valid data type (see Section 8.4.5).

The type `IF` may be any valid (user-defined) interface class.

*Limitation:* Arrays of ports are not supported. Structs containing ports are not supported.

### Signals and Channels

The only synthesisable channel is `sc_signal<T>` where the type `T` may be any valid data type (see Section 8.4.5).

The instantiation and naming requirements are exactly the same as in the case of ports (Section 8.4.3).
*Limitation:* Arrays of signals are not supported. Structs containing signals are not supported.

### Bindings

Every port of an instantiated module must be bound within the instantiating (parent) module.

Port bindings must be done via the `operator ()` as shown in the Listing 8.8.

```
#include <systemc.h>

SC_MODULE(Bottom)
{
  sc_in<bool>          p1;
  sc_out<int>          p2;
  sc_in<sc_uint<8> >   p3;
```

```
8  };
9
10 SC_MODULE(Middle)
11 {
12   sc_in<bool>          q1;
13   Bottom b;
14
15   sc_signal<int>  s;
16
17   SC_CTOR(Middle)
18   : b("b")
19   {
20     b.p1(q1);    // OK port-to-port binding
21     b.p2(s);     // OK port-to-signal binding
22                  // NOT OK: unbound b.p3
23   }
24 };
25
26 SC_MODULE(Top)
27 {
28   sc_in<sc_uint<8> >  r3;
29
30   Middle m;
31
32   SC_CTOR(Top)
33   : m("m")
34   {
35     m.b.p3(r3); // NOT OK: bypasses the module hierarchy
36   }
37 };
```

Listing 8.8: Synthesisable bindings

*Forbidden:* Bindings must not bypass modules within the hierarchy.

*Forbidden:* Positional binding is not allowed.

### 8.4.4 Processes

The following process types are supported:

- `SC_METHOD`
- `SC_CTHREAD`

Constraints on `SC_CTHREAD`s:

- `SC_CTHREAD`s must not share member variables of a module. If two processes must exchange data either use a signal or a Shared Object. If a data member is exclusively used by a single process, better make it a local variable of the process body.
- `SC_CTHREAD`s must not terminate, i.e. an infinite loop is required in the process body. If you somehow need a terminating `SC_CTHREAD`, place a `while(true) wait();` at the end of the process body.

Constraints on `wait(...)` usage (in `SC_CTHREAD`s):

- Arbitrary `wait(n)` is allowed, i.e. `n` can be any (integer) expression. More specifically it is not required that `n` can be determined at compile-time. *WARNING:* If `n` cannot be determined at compile- time it is the user's responsibility to ensure that `n` never becomes zero (If this happens during normal SystemC simulation, the SystemC kernel will raise an error). *Fossy* won't (can't) check this and consequently won't raise an error. As a workaround, you could write `if (n) wait(n);`. In this case, however, you have to take special care for the following rules, because the `wait()` is conditional in this case. The constraints on conditional `wait()`s, however, *are* checked by *Fossy*.
- Loops must either
  1. always run into a `wait` in each iteration
  2. or have a fixed number of iterations which can be determined at compile time (This restriction is not due to OSSS, but due to the following synthesis tool).

*Limitation: Fossy* can only determine the number of iterations of `for`-loops.

*Limitation:* `wait()`s in methods or functions are currently not allowed.

*Limitation:* `wait()`s in `switch` statements currently do not work.

Examples are shown in Listing 8.9

*Forbidden:* `sensitive_pos` and `sensitive_neg` are not allowed. Use the corresponding `.pos()` and `.neg()` methods of the port.

*Forbidden:* `watching(...)` is not allowed. Use `reset_signal_is(...)` instead.

```
1  #include <systemc.h>
2
3  SC_MODULE(Top)
4  {
5    sc_in<bool> clk,
6                reset;
7
8    sc_in<int>  px;
9
10   void myMethod();
11
12   // This CTHREAD is intended to show valid and invalid loop
13   // constructs (don't mind that the loops are actually unreachable).
14   void myCthread()
15   {
16     int x=4;
17     wait();
18
19     // Valid loop: always runs into a wait each iteration
20     while(true)
21     {
22       if (x--)
```

```
23          wait(1)
24        else
25          wait(2);
26      }
27
28      // Invalid loop: may do iterations without
29      // running into the wait;
30      while(true)
31      {
32        if (x--) wait();
33      }
34
35      // Valid loop: always runs into a wait each iteration
36      // Note that the wait() be skipped entirely if the condition
37      // is false before starting the loop.
38      while(x--) wait();
39
40      // Valid loop: always runs into a wait each iteration
41      while(true)
42      {
43        if (x--) wait();
44
45        "Some code";
46        wait();
47        "More code";
48      }
49
50      if(x==0) x=1;
51
52      // OK: some x which is not zero
53      wait(x);
54
55      // Valid loop: always runs into a wait each iteration...
56      while(true)
57      {
58        if (x--) wait();
59
60        // ... since this loop always causes a wait();
61        for (int i=0;i<3;++i)
62        {
63          if (i==x-1) wait(); else wait();
64        }
65      }
66
67      // Valid loop: always runs into a wait each iteration
68      do
69      {
70        wait();
71      } while (x--);
72
73    }
74
75    SC_CTOR(Top)
76    {
77      SC_METHOD(myMethod);
78      sensitive << clk.pos()     // Note: .pos()/.neg() only works
```

```
79                << reset;          //        for Boolean ports
80
81      sensitive << px;             // OK: there may be multiple
82                                   //      sensitives
83
84      SC_CTHREAD(myCthread(), clk.pos());
85      reset_signal_is(reset, true);
86    }
87  };
```

Listing 8.9: Synthesisable processes

### Effect of `wait()` usage on the number of states

In general the number of states of the resulting state machine equals the number of `wait()`s in the process body. A special case are loops: if there is a path through the loop body which does not contain any `wait()`s, *Fossy* tries to unroll that loop. Consequently the number of resulting states is the number of `wait()` statements[2] times the number of iterations. Please keep in mind that loops with many iterations which have to be unrolled due to non-optimal `wait()` usage, cause long synthesis runtimes, high memory consumption and finally a huge (but fast) circuit.
Another special case regarding the number of resulting states are `wait(n)`s. There are two different synthesis strategies. The first one is to simply unroll the `wait(n)` by `n` `wait(1)`s which results in `n` states. The second one is to replace the `wait(n)` by a loop with an unconditional `wait(1)` in the loop body. The resulting state machine of the second approach adds only one state to the state machine and one additional counter. The decision which of both approaches is chosen, depends on whether `n` is a compile-time known constant and also on its value.

### Reset

The reset signal of an `SC_CTHREAD` is determined by the correpsonding `reset_signal_is(..)` statement in the constructor body. Please note that the reset of an `SC_CTHREAD` is always synchronous to the `SC_CTHREAD`'s clock. Consequently, *Fossy* always creates a state machine with a synchronous reset.
Note that the pre-reset behaviour before and after synthesis may differ. This is due to the fact the in the simulation an `SC_CTHREAD` always starts from the beginning of the method, whereas after synthesis (and in real hardware) the register which encodes the current state will start with a random value until it is reset. Starting with a random state in the context of an `SC_CTHREAD` would mean to start at some arbitrary `wait()`. In other words, the SystemC simulation with `SC_CTHREAD`s suggests that the state machine starts in its initial state even without reset, which is generally not the case in real hardware. However, after a reset the state machines before and after synthesis show exactly the same behaviour.
Since an `SC_CTHREAD` begins its execution from the beginning of the method body when the reset signal becomes activated, the reset state is determined by path(s) from the beginning of the method body to all potential first[3] `wait()`s. Note that it is not necessary to test the

[2] In this case the `wait()`s will be conditional ones, because otherwise the loop would not have a path through its body without encountering a `wait()`.

[3] There may be more than one possible first `wait()` in the case of conditional `wait()`s in the reset part.

reset signal in the body of the process. A typical structure is shown in Listing 8.10. Note that it is also valid to move the first `wait()` into the `while(true)` loop, because *Fossy* will detect that the loop will always be entered. Consequently it is possible to safe one `wait()` and hence one state as shown in `myCthread2()`. The first variant, i.e. `myCthread()` will result in two equivalent states, both performing `x += px`.

```
1  #include <systemc.h>
2
3  SC_MODULE(Top)
4  {
5    sc_in<bool> clk,
6                reset;
7
8    sc_in<int>  px;
9
10   void myMethod();
11
12   void myCthread()
13   {                //
14     int x=4;       // Reset Part
15     wait();        //
16                        | State1
17     while(true)        |       +--+
18     {                  |       |  |
19       x += px;       V       V  | State2
20       wait();                   |
21                                 |  |
22     }                           +--+
23   }
24
25   void myCthread2()
26   {                //
27     int x=4;       // Reset Part
28                    //
29     while(true)    //  +--+
30     {              //  V  |
31       wait();      //     | State1
32       x += px;           |  |
33     }                    +--+
34   }
35
36   SC_CTOR(Top)
37   {
38     SC_CTHREAD(myCthread(), clk.pos());
39     reset_signal_is(reset, true);
40     SC_CTHREAD(myCthread2(), clk.pos());
41     reset_signal_is(reset, true);
42   }
43 };
```

Listing 8.10: Reset part of an `SC_CTHREAD`

### 8.4.5 Datatypes

The following basic data types can be used for writing synthesisable models:

- `bool`
- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`,
- `int`, `unsigned int`,
- `long`, `unsigned long`,
- `long long`, `unsigned long long`,
- `sc_int<N>`, `sc_uint<N>`,
- `sc_bigint<N>`, `sc_biguint<N>`,
- `sc_bv<N>`,
- Enumeration types,

Currently not supported are:

- `sc_bit`
- `sc_logic`
- `sc_lv<N>`,
- `sc_fixed<WL,IL, Q, O, n>`, `sc_ufixed<WL,IL, Q, O, n>`
- `sc_fixed_fast<WL,IL, Q, O, n>`, `sc_ufixed_fast<WL,IL, Q, O, n>`

Complex types like arrays, structures, classes and unions are synthesisable as long as they are constructed from synthesisable basic types. Classes, however, are regarded in greater detail in Section 8.4.7.

All data types mentioned above can be used in the following places:

- Local variables in functions and processes
- Member variables
- Function parameters and member function parameters
- Signals (`sc_signal<T>`). Note: This requires `T` to define the `operator==(...)` and an `operator<<(...)` for stream insertion (and a `sc_trace(...)` function)
- Signal-ports (`sc_in<T>`, `sc_out<T>`, `sc_inout<T>`)
- `typedef`

*Limitation:* The resolved signal `sc_signal_rv<N>` and the corresponding ports `sc_in_rv<N>`, `sc_out_rv<N>` and `sc_inout_rv<N>` are currently not allowed for synthesisable models.

### 8.4.6 Statements and Expressions

The basic statements of C/C++ such as variable declaration and definition, assignments, control structures like `if () else`, `switch`, `for` and `while` loops are synthesisable. It is allowed to have `switch` statements with fall-through cases, i.e. cases without a `break` statement. Functions and function calls are synthesisable as long as they operate on synthesisable data types and consist of synthesisable constructs.

Parameters may be passed by value or by reference and may be `const` or non-`const`.

*Forbidden:* Functions and operators must not return references.

*Limitation:* Chaining multiple `operator =` in assignments to ports or signals (such as `a=b=c=d=0;` is not supported, yet.

*Limitation:* A `case`-part of a `switch`-statement which solely contains a `break;` is not supported. Workaround: insert an empty expression statement (`;`) right after the case label: `switch(i) { case 1: ; break; ... }`.

*Limitation:* A variable declaration in a `switch`-block before the first `case` label is not supported.

*Limitation:* Non-toplevel `case` labels are not supported.

*Limitation:* Variable declarations in `switch` clauses without a surrounding block are not supported.

*Limitation:* A non-void routine must not contain a `return` statement without an expression.

*Limitation:* Recursion is not supported.

*Limitation:* `sizeof(...)` is not supported.

*Limitation:* Only SystemC bitvector literals with prefix `"0b0"` are supported, e.g. `sc_int<3> x; x="0b0111"; x="0b0" "111";`

### 8.4.7 Classes and Inheritance

The following features of classes are allowed for synthesis:

- Non-`const` non-`static` data members
- `const` non-`static` data members
- `const static` data members
- Non-`static` member functions

- `static` member functions
- Virtual member functions (in conjunction with polymorphic objects)
- Pure virtual member functions (in conjunction with polymorphic objects)
- Base class(es) [Limitation: no non-virtual multiple inheritance from one base]
- Virtual base class(es) [Limitation: the virtual bases must not have any data members]
- Constructors [Limitation: copy constructor must have exactly one `const&` argument; default constructor must not have defaulted arguments]
- Member initialiser list
- Overloaded operators
- User-defined implicit casts [Limitation: May collide with *Fossy*'s SystemC header files, especially the integer types]
- `explicit` constructors

*Limitation:* Copy assignment operators (`operator=`) must have the return type `void` and exactly one `const&` argument.

*Limitation:* If a user class contains an array member attribute, a copy constructor must be defined.

*Limitation:* Pointers to unused classes or template instances are not allowed. Note: This includes types like `sc_signal<>` which are actually templates.

Each data member must be of a synthesisable type (see Section 8.4.5) and member functions must follow the same restrictions as functions do (see Section 8.4.6).

*Forbidden:* Classes must not have an own thread of control, i.e. any `SC_METHOD`s, `SC_THREAD`s or `SC_CTHREAD`s.

### 8.4.8 Templates

Templates can be used to parameterise functions, classes/structs and member functions. Note that template classes may have template methods. Template specialisation and partial template specialisation are supported.

### 8.4.9 Namespaces

Namespaces are supported. The namespaces `osss`, `osss::synthesisable`, `sc_dt`, `sc_core` and `std` are reserved and must not be extended by the user.

### 8.4.10 Polymorphic Objects

*Limitation:* Polymorphic objects are not yet supported.

### 8.4.11 Shared Objects

Each guarded method must overwrite a virtual method from its interface class.

Parameters of guarded methods must be passed by value or const reference.

A guarded method which does not write any attribute must have a `wait()` in its body.

*Limitation:* The `schedule()` method of a scheduler must have a single `return` at the end of its body.

*Limitation:* Initialisation of schedulers must be performed in the constructor body, i.e. initialiser lists are currently ignored. This limitation also applies to all inherited constructors.

### 8.4.12 Non-Synthesisable

The following C++/SystemC/OSSS constructs are not synthesisable:

- OSSS architecture layer models
- Pointers (**Exception:** instantiation of modules, port and signals)
- Pointer arithmetics
- Member pointers
- Dynamic memory allocation with `new` and `delete`
- Placement-`new`
- Exceptions, throw-specifications
- Runtime `typeid`
- Reference types (**Exception:** function parameters)
- `dynamic_cast`
- Destructors (except for empty destructors)
- Static data members
- `mutable`, `volatile`
- `auto`, `register`
- `asm`

- `goto`
- `inline` (has no effect, does not harm)
- Standard C/C++ libraries with string handling, file I/O, ...
- Floating point arithmetic
- Bit fields (not needed – use SystemC data types instead)
- `friend` (partially implemented)
- `sc_time` (partially implemented)

# 9 Summary and Support

## 9.1 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.

## 9.2 Conclusion

The basic features and modelling techniques of OSSS, including modelling of dynamically reconfigurable digital hardware, have been introduced in this document. More information can be found within the OSSS source code package.
The presented library targets flexible hardware/software communication for different target architectures and dynamic partial reconfiguration of digital hardware components.

# Appendix A Examples of the OSSS 2.2.0 Simulation Library

The examples chapter aims to illustrate the application of the previously described modelling elements.

## A.1 OSSS-Channel examples

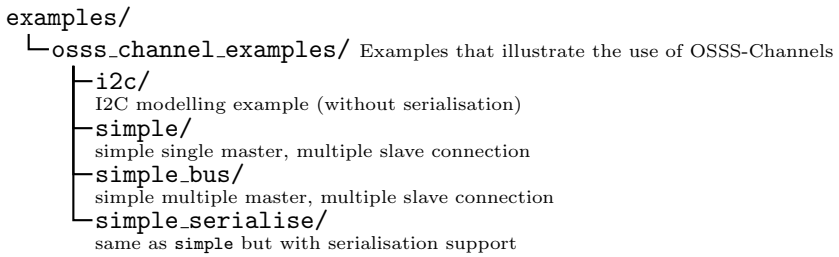



Figure A.1: Structure of the OSSS 2.2.0 simulation library `osss_channel_examples` directory

## A.2 OSSS Methodology examples

examples/
- osss_osss_examples/ Explains how the Application Layer Model is mapped to the Virtual Target Architecture Layer, where communication is performed by RMI (Remote Method Invocation) through OSSS-Channels
  - bridge/ A Xilinx OPB to ARM APB bridge illustrating the mapping from the *Application Layer* to *Virtual Target Architecture Layer*
  - producer_consumer/ A buffer (FIFO modelled by a Shared Object) of fixed/bounded size. A producer writes data packets into the buffer while a consumer reads them from the buffer.
    - extended/ This version consists out of a producer SW Task a buffer Shared Object and two HW consumer processes, each of them in a module of their own. This version can be used in different configurations on the Virtual Target Architecture Layer, thus demonstration the flexibility for architecture exploration. This version enables some kind of packet throughput measurements for the comparison of different architecture mappings.
    - simple/ This simple version consists out of a two processes (A producer and a consumer, both implemented as HW modules) and a buffer Shared Object.
  - raytracer/ A simple SW raytracer. This example demonstrates multiple SW Tasks accessing a single Shared Object. Two SW Tasks perform raytracing of a pre-defined scene; one SW Task traces the upper part of the scene while the other one traces the lower part. These Tasks write their tracing results to a Shared Object. A third SW Task works as a monitor that waits until the scene has been traced completely, reads the results from the Shared Object and outputs them as a bitmap image.
  - shared_alu/ Contains an ALU that is shared between two client processes. The ALU is modelled by a Shared Object. The client processes are performing a square-root calculation using the Newton method.
  - sw_timing/ Example for the usage of an `osss_software_task` and in conjunction with Estimated Execution Times (EETs) & Required Execution Times (RETs)
  - transceiver/ A parallel to serial data converter illustrating the mapping from the *Application Layer* to *Virtual Target Architecture Layer*

Figure A.2: Structure of the OSSS 2.2.0 simulation library `osss_osss_examples` directory

## A.3 OSSS+R examples

For the introduction of the OSSS+R modelling elements of OSSS we have chosen a reconfigurable audio player application. Figure A.3 shows the structure of the OSSS 2.2.0 simulation library `osss_recon_examples` directory.

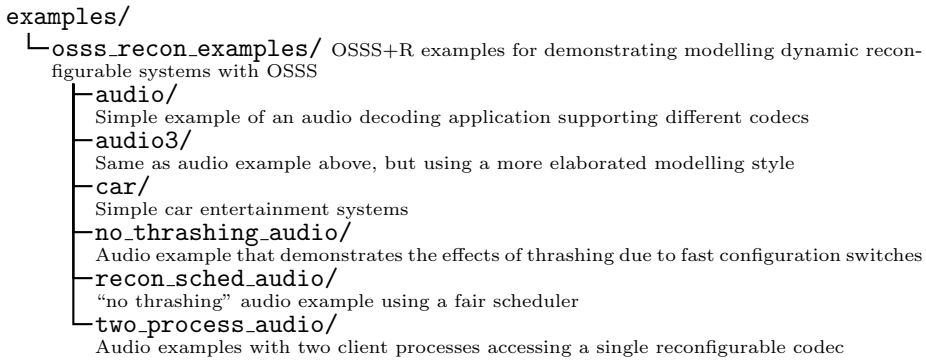



Figure A.3: Structure of the OSSS 2.2.0 simulation library `osss_recon_examples` directory

We start with the introduction of the reconfigurable audio player by using a "plain" Recon Object in a pure hardware design. This is followed by the application of the Mediator Pattern by introducing a Software Task that communicates with the audio player Recon Object HW implementation and triggers reconfiguration. After the introduction of hardware/software communication we extend the audio player example by Named Contexts that enable more advanced features for reconfiguration. The example chapter closes with the application of the configuration prefetching pattern that is based on Named Contexts and Locks.

### A.3.1 Reconfigurable Audio Player

In this example we are designing a simple reconfigurable audio player that is capable of encoding and decoding MP3, OggVorbis, and AAC data. Some of the code presented has been modified or shortened for readability and simplicity. The complete source code of this example is included with the OSSS installation package.
The starting point for the design is the class hierarchy given in Figure A.4. There is a base class `AudioCodec` that defines methods for input and output buffer handling. Moreover, it contains the abstract methods `encode()` and `decode()`. All three derived classes: `MP3Codec`, `AACCodec` and `OggVorbisCodec` implement `encode()` and `decode()` to their designated codecs.

#### Defining Classes

Listing A.1 shows the definition of the audio player base class `AudioCodec`[1].

```
1 template<class BufferType>
2 class AudioCodec : public osss::osss_object {
3 protected:
```

[1]The template parameter *BufferType* may be any class that implements the methods `Dataword read()`, `void write(Dataword d)`, `bool full()` and `bool empty()`.

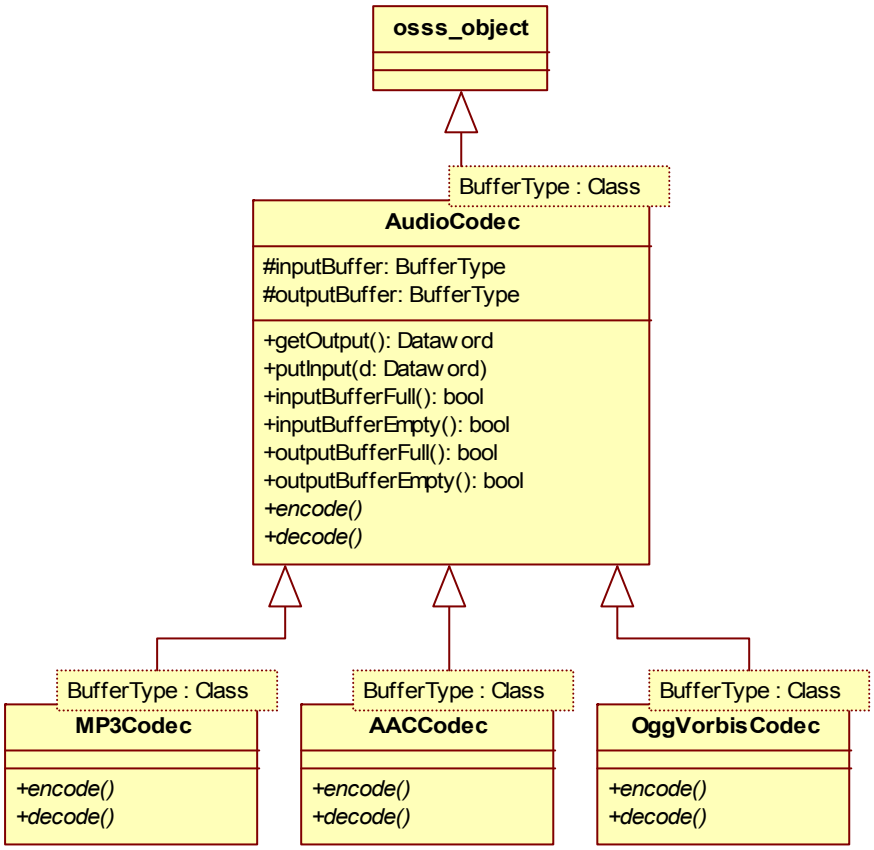



Figure A.4: Class hierarchy for audio example

```
4   BufferType ibuf, obuf;
5
6  public:
7   AudioCodec() { OSSS_BASE_CLASS( osss::osss_object ); }
8
9   virtual void encode() {};
10  virtual void decode() {};
11
12  void putInput(Dataword d) { ibuf.write(d); }
13  bool inputBufferEmpty() { return ibuf.empty(); }
14  bool inputBufferFull() { return ibuf.full(); }
15
16  Dataword getOutput() { return obuf.read(); }
17  bool outputBufferEmpty() { return obuf.empty(); }
18  bool outputBufferFull() { return obuf.full(); }
19
20 protected:
21  Dataword getInput() { return ibuf.read(); }
22  void putOutput(Dataword d) { obuf.write(d); }
23 };
```

Listing A.1: Definition of audio player base-class

### Implementing Methods

Listing A.2 shows the definition of the three different codec implementations.

```
1  template<class BufferType>
2  class MP3Codec : public AudioCodec<BufferType> {
3  public:
4    typedef AudioCodec<BufferType> base_type;
5
6    MP3Codec() { OSSS_BASE_CLASS(base_type); }
7
8    virtual void encode() {
9      while ( ! base_type::inputBufferEmpty()
10           && ! base_type::outputBufferFull()) {
11       Dataword tmp = base_type::getInput();
12       // perform mp3 encoding ...
13       base_type::putOutput( encodedData );
14     }
15   }
16
17   virtual void decode() {
18     while ( ! base_type::inputBufferEmpty()
19           && ! base_type::outputBufferFull()) {
20       Dataword tmp = base_type::getInput();
21       // perform mp3 decoding ...
22       base_type::putOutput( decodedData );
23     }
24   }
25 };
26
27 template<class BufferType>
28 class AACCodec : public AudioCodec<BufferType> {
29 public:
30   typedef AudioCodec<BufferType> base_type;
31   AACCodec() { OSSS_BASE_CLASS(base_type); }
32
33   virtual void encode() { /* perform AAC encoding ... */ }
34   virtual void decode() { /* perform AAC decoding ... */ }
35 };
36
37 template<class BufferType>
38 class OggVorbisCodec : public AudioCodec<BufferType> {
39 public:
40   typedef AudioCodec<BufferType> base_type;
41   OggVorbisCodec() { OSSS_BASE_CLASS(base_type); }
42
43   virtual void encode() { /* perform OggVorbis encoding ... */ }
44   virtual void decode() { /* perform OggVorbis decoding ... */ }
45 };
```

Listing A.2: Definition of audio player classes

In this example there are the methods `decode()` and `decode()` which are implemented in every class. The implementation of `decode()` and `encode()` in `MP3Player` is given in Listing A.2. For simplicity, the methods do not implement the decoding and encoding algorithms but generate a debug message and does simple data manipulation.

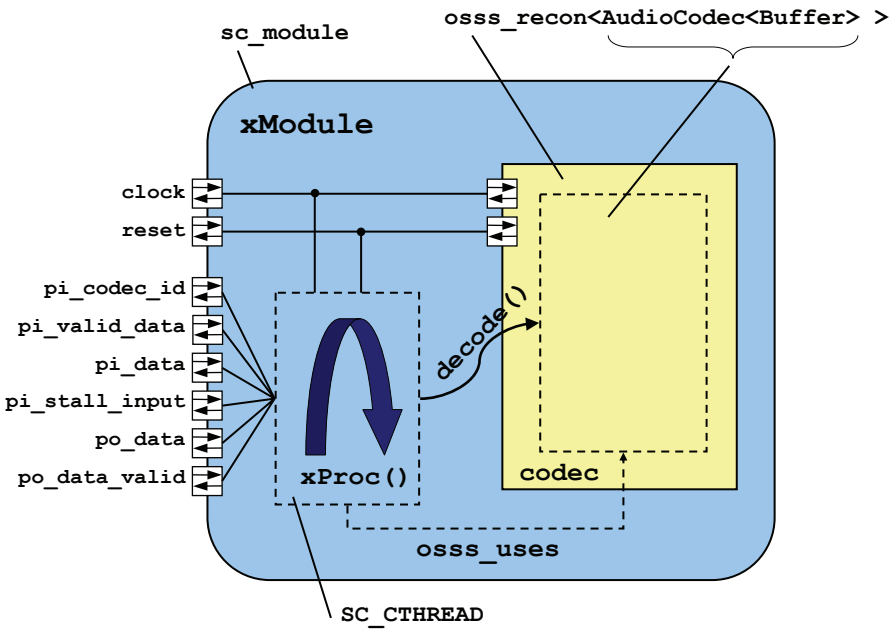



Figure A.5: Structure of reconfigurable audio player

### Building Structure

The structure of a system is described in OSSS+R using standard SystemC elements like modules, ports and signals and OSSS+R specific elements like the Recon Object. This example consists of a module that contains one process one Recon Object, and clock and reset ports.
The definition of the audio player module is given in Listing A.3.

```
1  SC_MODULE(xModule) {
2    sc_in< bool > clock;
3    sc_in< bool > reset;
4
5    sc_in< int >        pi_codec_id;
6    sc_in< Dataword >   pi_data;
7    sc_in< bool >       pi_data_valid;
8    sc_out< bool >      po_stall_input;
9
10   sc_out< Dataword > po_data;
11   sc_out< bool >      po_data_valid;
12
13   osss::osss_recon< AudioCodec<Buffer> > codec;
14 protected:
15   void flush(CodecID);
16   void xProc();
17 public:
18   SC_CTOR(xModule) : codec("codec") {
19     // connect Recon Object with clock & reset
20     codec.clock_port(clock);
21     codec.reset_port(reset);
22
23     SC_CTHREAD(xProc, clock.pos());
24     reset_signal_is(reset, true);
```

```
25      // register use of codec Recon Object with xProc process
26      osss_uses(codec);
27    }
28  };
```

Listing A.3: Definition of audio player module

Line 14 defines the Recon Object `codec` with `AudioCodec<Buffer>` as the interface parameter. The following lines (21) define the constructor of `xModule`. The clock and reset ports of `codec` are bound to the corresponding ports of `xModule`. The method `xProc` is declared to be a `SC_CTHREAD` with `clock` and `reset` as its control signals. The `osss_uses`-statement registers the process `xProc` to the Recon Object `codec`.

**Using the Recon Object** Listing A.4 shows parts of the definition of the user process `xProc()`. This example does not use Named Contexts. Here, the objects are assigned directly to the Recon Object. Every assignment of a new object results in the destruction of the previous one and the creation of a Temporary Context within the Recon Object. Depending on the assignment of the `pi_in_codec_id` signal the `xProc()` process consecutively assigns an `MP3Codec` (line 28), an `AACCodec` (line 30), and an `OggVorbisCodec` (line 32) object to `codec`. The `decode()` method is called in the `flush()` method, before a context switch (line 25) and after the context switch has been performed (line 38).

```
1  void xModule::flush(CodecID current_codec_id) {
2    if (current_codec_id != NO_ID) {
3      codec->decode(); // call decode on Recon Object
4      while (! codec->outputBufferEmpty() ) {
5        po_data.write( codec->getOutput() );
6        po_data_valid.write( true );
7        wait();
8        po_data_valid.write( false );
9      }
10   }
11 }
12
13 void xModule::xProc() {
14   CodecID current_codec_id = NO_ID;
15   po_stall_input.write( true );
16   po_data_valid.write( false );
17   while (true) {
18     po_stall_input.write( true );
19
20     bool codec_change = (pi_codec_id.read() != current_codec_id);
21     bool no_input     = (pi_data_valid.read() == false);
22     if (codec_change || no_input)       {
23       flush(current_codec_id);
24
25       // New codec
26       if (codec_change && (pi_codec_id.read() != NO_ID))
27       {
28         switch( pi_codec_id.read() ) // switch to next codec
29         { case MP3_ID : codec = MP3Codec<Buffer>();        break;
30           case AAC_ID : codec = AACCodec<Buffer>();        break;
31           case OGG_ID : codec = OggVorbisCodec<Buffer>(); break;
32         }
```

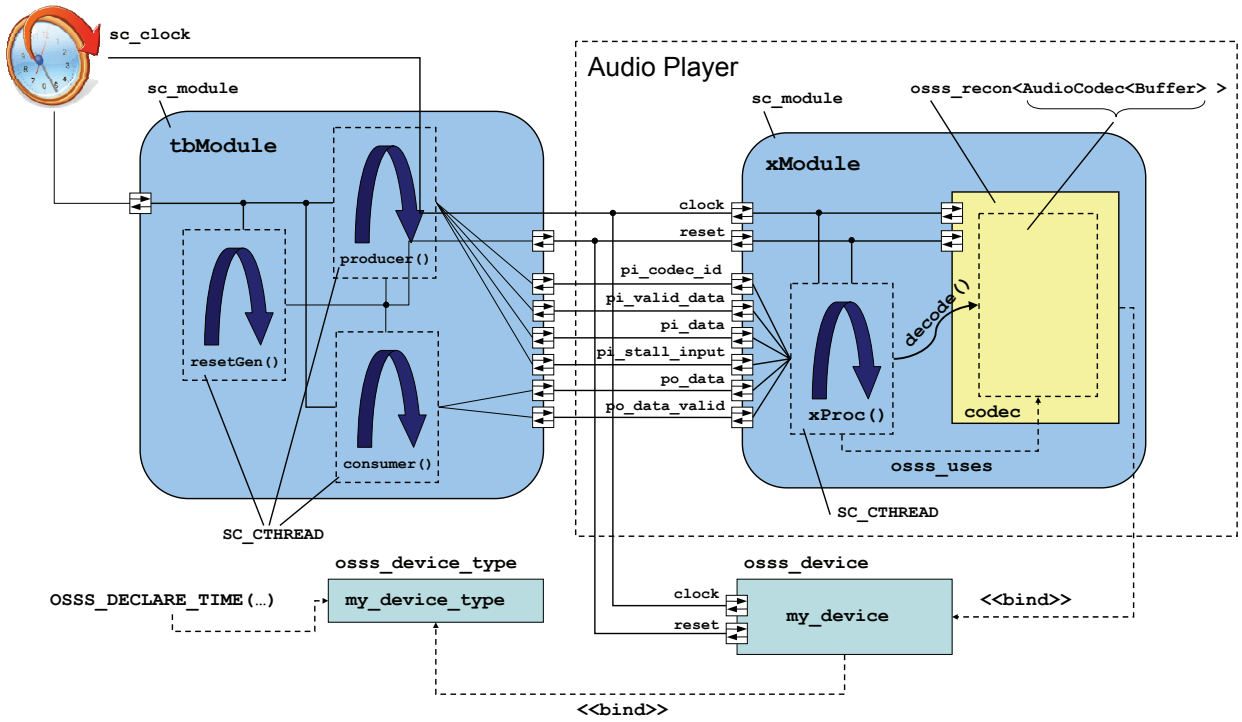



Figure A.6: Structure of entire reconfigurable audio player (including testbench)

```
33        }
34        wait();
35        current_codec_id = pi_codec_id.read();
36      } else { // no codec change and valid input
37        if ( ! codec->inputBufferFull() ) {
38          po_stall_input.write( false );
39          Dataword d = po_data.read();
40          wait();
41          po_stall_input.write( true );
42          codec->putInput( d );
43        } else {
44            wait();
45        }
46      }
47    } // while
48  }
```

Listing A.4: Definition of audio player module process

**Creating a Testbench** For simulating the design some additional testbench components are needed. The `xModule` module is instantiated together with a clock and reset signal within the `sc_main` function. Every design that uses one or more Recon Objects needs at least one device object. The overall structure, including the testbench, of the reconfigurable audio player is shown in Figure A.6.

Listing A.5 shows one possible definition of an OSSS+R test scenario. Line 15 defines the FPGA device `my_device_type`. The following lines assign reconfiguration times to the combinations of device type and configuration object. Line 29 instantiates one device object

my_device of the previously defined device type my_device_type. Line 31 binds the codec Recon Object to my_device. The following lines bind the clock and reset ports of my_device to corresponding signals.

```
1  int sc_main(int /*argc*/, char * /*argv*/[])
2  {
3    // OSSS+R requires and explicit use of class sc_clock
4    // for the global system clock
5    sc_clock clock("main_clock", sc_time(10, SC_NS));
6    sc_signal<bool> reset;
7
8    xModule dut("design_under_test");
9    tbModule tb("testbench_module");
10
11   // ...
12
13   // We declare that we have a special type of FPGA. This is
14   // a type declaration (like "Xilinx Virtex II 1000-4").
15   osss::osss_device_type my_device_type("My FPGA device type");
16
17   // Specify, how the logic copy times (unused in this example,
18   // so please ignore the 26us entries) and reconfiguration times
19   // are, if a certain class (e.g. OggVorbisPlayer<Memory> is to
20   // be configured to the FPGA.
21   OSSS_DECLARE_TIME(my_device_type, AudioCodec<Buffer>,
22                     sc_time(26, SC_US), sc_time(200, SC_US));
23   OSSS_DECLARE_TIME(my_device_type, MP3Codec<Buffer>,
24                     sc_time(26, SC_US), sc_time(200, SC_US));
25   OSSS_DECLARE_TIME(my_device_type, AACCodec<Buffer>,
26                     sc_time(26, SC_US), sc_time(300, SC_US));
27   OSSS_DECLARE_TIME(my_device_type, OggVorbisCodec<Buffer>,
28                     sc_time(26, SC_US), sc_time(250, SC_US));
29   osss::osss_device my_device(my_device_type, "my_device");
30
31   dut.codec( my_device );
32
33   my_device.clock_port(clock);
34   my_device.reset_port(reset);
35
36   sc_start(); // start the simulation
37
38   return EXIT_SUCCESS;
39 }
```

Listing A.5: Definition of *sc_main* for the audio player

### Running the Example

The example can be compiled and simulated like SystemC designs. However, one has to make sure that the correct options are set to include the OSSS+R header files and to link against the OSSS+R library. The installation package includes example Makefiles which can easily be edited to fit other designs. Detailed information is given in the README file that comes with the OSSS+R package.

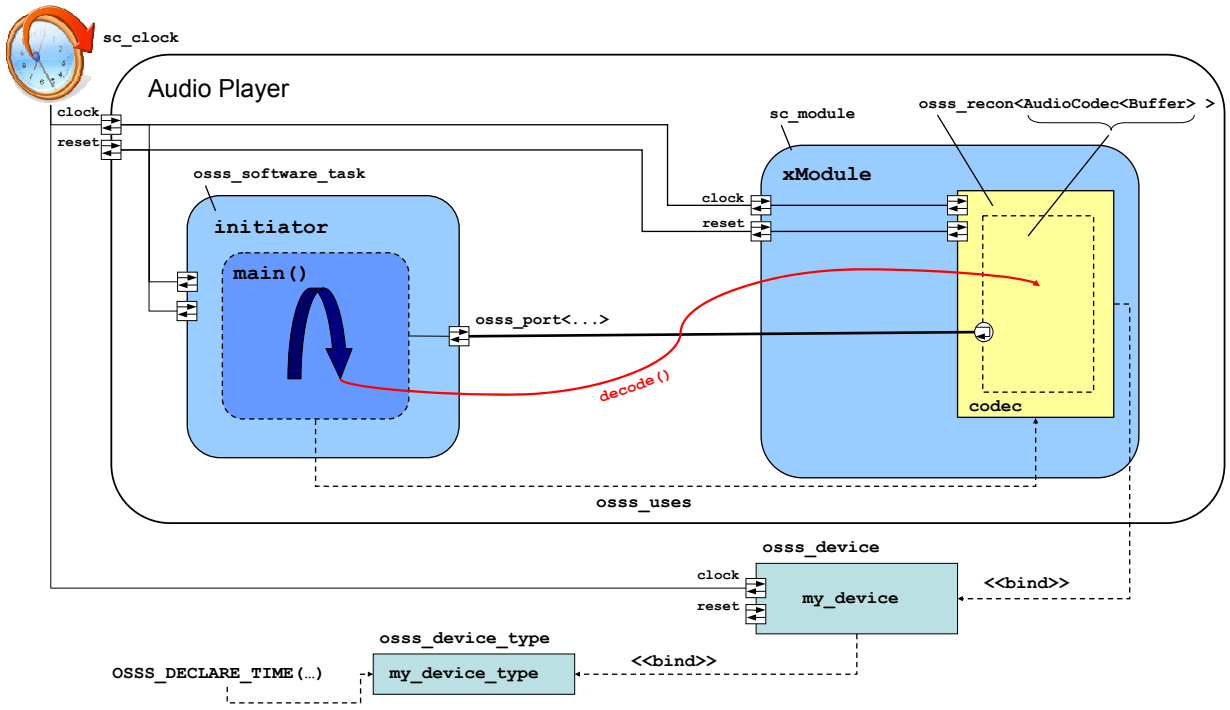



Figure A.7: Structure of reconfigurable audio player using Software Task

## A.3.2 Reconfigurable Audio Player using Mediator Pattern

In this section we demonstrate how to apply the Mediator Pattern from Section 7.1 to the reconfigurable audio player introduced in Section A.3.1.

### Overview

As shown in Figure A.7 we extend the reconfigurable audio player from Figure A.6 by a Software Task. In our example the Software Task replaces the previously used testbench module. In the presented scenario we aim to call all methods on the `codec` Recon Object directly within the main process of the Software Task. Moreover, we want to trigger class switches, i.e. reconfiguration, directly from the Software Task.
In the following subsections we demonstrate how this scenario can be set up using the Mediator Pattern.

### Application Layer Model

We start with the description of the Application Layer Model. This subsection describes all the steps that are necessary to apply the Mediator Pattern to the reconfigurable audio player.

**Defining Abstract Base Class** In a first step we need to extend the audio codec class hierarchy from Figure A.4 by an interface class. This interface needs to define all methods that should be callable through the mediator.
Figure A.8 shows the extension of the class hierarchy by the `AudioCodec_if` interface class. Listing A.6 defines this pure virtual interface class.

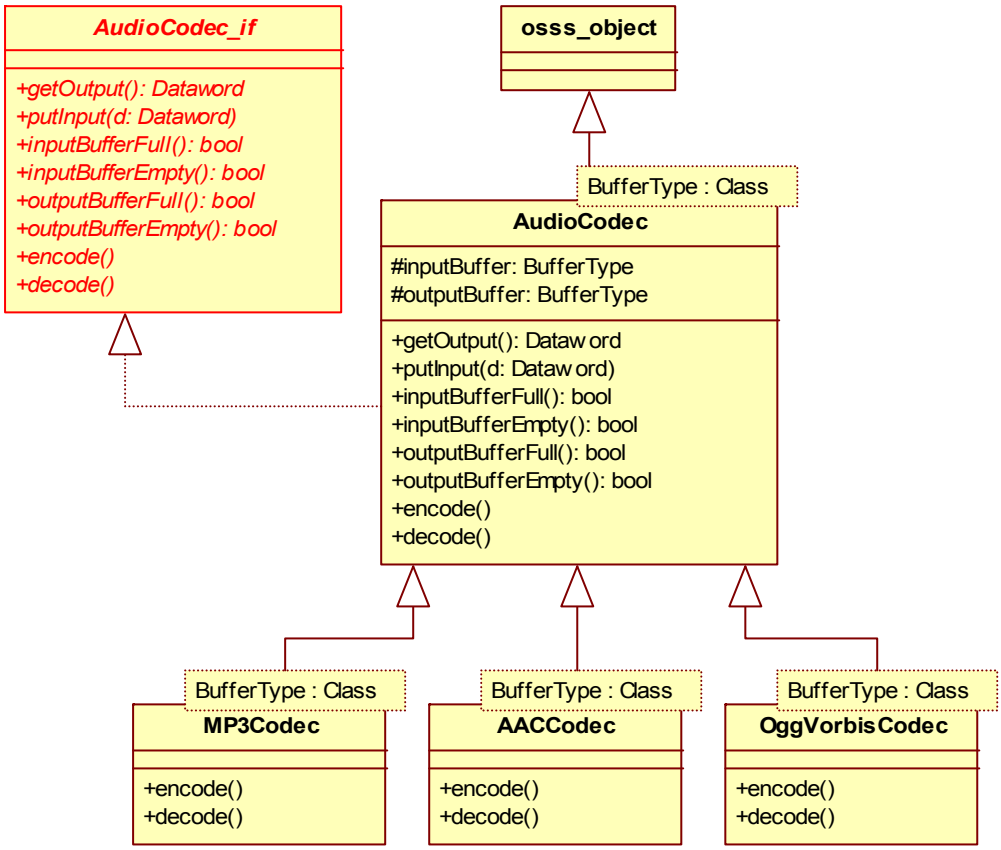



Figure A.8: Interface extended class hierarchy from Figure A.4

```
1  class AudioCodec_if {
2   public:
3    virtual void encode() = 0;
4    virtual void decode() = 0;
5
6    virtual Dataword getOutput() = 0;
7    virtual void putInput(Dataword d) = 0;
8    virtual bool inputBufferEmpty() = 0;
9    virtual bool inputBufferFull() = 0;
10   virtual bool outputBufferEmpty() = 0;
11   virtual bool outputBufferFull() = 0;
12 };
```

Listing A.6: Definition of the pure virtual audio interface class

**Defining Mediator Types** To make the usage of the Mediator Pattern a little bit more convenient the OSSS library provides some helper macros. Each Mediator Pattern is build for its unique interface class. In this example the pattern is build for the `AudioCodec_if` interface class. In a first step the designer needs to register of all types involved:

- Class types used for reconfiguration (MP3, AAC, OggVorbis)
- Method types used for communication (all method from the interface class)

```
1  OSSS_MEDIATOR_TYPES_FOR( AudioCodec_if ) {
2    OSSS_MEDIATOR_CLASS_TYPES_BEGIN
3      OSSS_MEDIATOR_REGISTER_CLASS_TYPE( mp3codec )
4      OSSS_MEDIATOR_REGISTER_CLASS_TYPE( aaccodec )
5      OSSS_MEDIATOR_REGISTER_CLASS_TYPE( vorbiscodec )
6    OSSS_MEDIATOR_CLASS_TYPES_END
7
8    OSSS_MEDIATOR_METHOD_TYPES_BEGIN
9      OSSS_MEDIATOR_REGISTER_METHOD_TYPE( encode )
10     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( decode )
11     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( getOutput )
12     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( putInput )
13     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( inputBufferEmpty )
14     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( inputBufferFull )
15     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( outputBufferEmpty )
16     OSSS_MEDIATOR_REGISTER_METHOD_TYPE( outputBufferFull )
17   OSSS_MEDIATOR_METHOD_TYPES_END
18 };
```

Listing A.7: Definition of types for the Meditor Pattern

Listing A.7 registers these types for the `AudioCodec_if` interface class used in the reconfigurable audio player. Moreover, it registers all three audio codec classes. For class type registration a unique alias name has to be chosen (e.g. mp3codec for class `MP3Codec<Buffer>`), and for method type registration the exact name of the method (as defined in `AudioCodec_if`) needs to be used.

**Defining Initiator & Target Interfaces** The next step is the definition of the initiator and the target interface classes of the Mediator Shared Object. As shown in Listing A.8 and Listing A.9 each method signature of the interface class needs to be written in OSSS "standard macro" style. This style has been introduced for guarded methods of Shared Objects for the first time.
Please note that the notion of no method arguments can either be `OSSS_PARAMS(0)` or `OSSS_VOID`.

```
1  OSSS_MEDIATOR_INITIATOR_IF_FOR( AudioCodec_if ) {
2    OSSS_MEDIATOR_INITIATOR_IF_CTOR( AudioCodec_if );
3
4    OSSS_MEDIATOR_INITIATOR_IF_METHOD_VOID( encode, OSSS_VOID );
5    OSSS_MEDIATOR_INITIATOR_IF_METHOD_VOID( decode, OSSS_VOID );
6    OSSS_MEDIATOR_INITIATOR_IF_METHOD( Dataword, getOutput, OSSS_VOID );
7    OSSS_MEDIATOR_INITIATOR_IF_METHOD_VOID( putInput,
8                                            OSSS_PARAMS(1, Dataword, d) );
9    OSSS_MEDIATOR_INITIATOR_IF_METHOD( bool, inputBufferEmpty, OSSS_VOID );
10   OSSS_MEDIATOR_INITIATOR_IF_METHOD( bool, inputBufferFull, OSSS_VOID );
11   OSSS_MEDIATOR_INITIATOR_IF_METHOD( bool, outputBufferEmpty, OSSS_VOID );
12   OSSS_MEDIATOR_INITIATOR_IF_METHOD( bool, outputBufferFull, OSSS_VOID );
13 };
```

Listing A.8: Definition of initiator interface

```
1  OSSS_MEDIATOR_TARGET_IF_FOR( AudioCodec_if ) {
2    OSSS_MEDIATOR_TARGET_IF_CTOR( AudioCodec_if );
3
```

```
4   OSSS_MEDIATOR_TARGET_IF_METHOD_VOID( encode, OSSS_VOID) );
5   OSSS_MEDIATOR_TARGET_IF_METHOD_VOID( decode, OSSS_VOID) );
6   // ... other methdos from AudioCodec_if ...
7 };
```

Listing A.9: Definition of target interface

**Defining the Mediator Shared Object** The definition of the Mediator Shared Object itself requires the definition of the mediator types, initiator and target interfaces. When applying this pattern the following sequence of definitions is required:

1. Mediator types
2. Initiator & target interface
3. Mediator Shared Object

Listing A.10 defined the Mediator Shared Object for the `AudioCodec_if` interface class. It uses the same macros as the initiator and target interfaces.

```
1 OSSS_MEDIATOR_CLASS_FOR( AudioCodec_if ) {
2   OSSS_MEDIATOR_CLASS_CTOR( AudioCodec_if );
3
4   OSSS_MEDIATOR_METHOD_VOID( encode, OSSS_VOID) );
5   OSSS_MEDIATOR_METHOD_VOID( decode, OSSS_VOID );
6   // ... other methdos from AudioCodec_if ...
7 };
```

Listing A.10: Definition of Mediator Shared Object

**Defining the Mediator Transactor Module** The Mediator Transactor Module includes the Recon Object and a process that uses the target interface of the Mediator Shared Object for performing requested method calls directly on the Recon Object. Moreover, it performs the requested class switches (reconfiguration).
Listing A.11 shows the definition of the Mediator Transactor Module `xModule`. For accessing the Mediator Shared Object is has an `osss_port` that connects to the `osss_mediator_target_if` (line 5 & 6). It includes the `codec` Recon Object (line 8) and the module constructor (line 10) performs clock and reset port bindings, and the registration of the transactor process `xProc` at the Recon Object (line 19).

```
1  SC_MODULE( xModule ) {
2    sc_in<bool> clock;
3    sc_in<bool> reset;
4
5    osss::osss_port<osss_shared_if<
6      osss_mediator_target_if< AudioCodec_if > > > mediator_so_port;
7
8    osss::osss_recon< AudioCodec<Buffer> > codec;
9
10   SC_CTOR( xModule ) :
11     mediator_so_port("mediator_so_port"),
12     codec("codec")
13   {
14     codec.clock_port(clock);
15     codec.reset_port(reset);
```

```
16
17      SC_CTHREAD(xProc, clock.pos());
18      reset_signal_is(reset, true);
19      osss::osss_uses(codec);
20    }
21
22  protected:
23    OSSS_HAS_MEDIATOR_TRANSACTOR_PROCESS_FOR( AudioCodec_if );
24
25    OSSS_MEDIATOR_TRANSACTOR_PROCESS( xProc, codec, MP3Codec<Buffer> )
26    {
27      OSSS_MEDIATOR_TRANSACT_SWITCH_BEGIN( mediator_so_port, codec)
28        // class switches
29        OSSS_MEDIATOR_TRANSACT_CLASS( mediator_so_port, codec,
30                                      MP3Codec<Buffer>, mp3codec )
31        OSSS_MEDIATOR_TRANSACT_CLASS( mediator_so_port, codec,
32                                      AACCodec<Buffer>, aaccodec )
33        OSSS_MEDIATOR_TRANSACT_CLASS( mediator_so_port, codec,
34                                      OggVorbisCodec<Buffer>, vorbiscodec )
35        // method calls
36        OSSS_MEDIATOR_TRANSACT_METHOD_VOID( mediator_so_port, codec,
37                                            encode, OSSS_VOID )
38        OSSS_MEDIATOR_TRANSACT_METHOD_VOID( mediator_so_port, codec,
39                                            decode, OSSS_VOID )
40        // ... other methdos from AudioCodec_if ...
41        OSSS_MEDIATOR_TRANSACT_METHOD( mediator_so_port, codec,
42                                       Dataword, getOutput , OSSS_VOID )
43      OSSS_MEDIATOR_TRANSACT_SWITCH_END
44    }
45  };
```

Listing A.11: Definition of Mediator Transactor Module

A Mediator Transactor Process needs to be defined for each interface class. For each transactor process that is implemented inside a module the `OSSS_HAS_MEDIATOR_TRANSACTOR_PROCESS_FOR` macro is required. A transactor process definition starts with the `OSSS_MEDIATOR_TRANSACTOR_PROCESS` macro that has three parameters:

1. process name,
2. Recon Object name,
3. and initial configuration of the Recon Object.

The process body starts with the `OSSS_MEDIATOR_TRANSACT_SWITCH_BEGIN` macro with two parameters:

1. Mediator Shared Object port name
2. and Recon Object name.

This block needs to be terminated with the `OSSS_MEDIATOR_TRANSACT_SWITCH_END` (line 43) macro.
Between the `_SWITCH_BEGIN` and `_SWITCH_END` macros all supported class type switches and method calls to the Recon Object need to defined. Class switches are defined by using the `OSSS_MEDIATOR_TRANSACT_CLASS` macro with the following parameters:

1. Mediator Shared Object port name,

2. Recon Object name,
3. Recon Object content class type,
4. and the previously user-defined type alias.

These macros map a Recon Object content class to the user-defined type alias. In listing Listing A.11 we describe the following mapping:

- `MP3Codec<Buffer>` → mp3codec (line 29 & 30)
- `AACCodec<Buffer>` → aaccodec (line 31 & 32)
- `OggVorbisCodec<Buffer>` → vorbiscodec (line 33 & 34)

Method calls to the Recon Object are defined using the `OSSS_MEDIATOR_TRANSACT_METHOD` (for methods with a return type) or `OSSS_MEDIATOR_TRANSACT_METHOD_VOID` (for methods with no return type) macros. The first and the second parameter are the same as for the class switch macro, and the remaining parameters are the same as for the OSSS Mediator initiator and target interface macros.

**Defining the Software Task** The Software Task is the initiator of the Mediator Pattern. Listing A.12 defines the software task for this example. Like the transactor module on the hardware side, the software task has an `osss_port` (line 2-4) to access the Mediator Shared Object. This port connects to the `osss_mediator_initiator_if` interface.

```
1  OSSS_SOFTWARE_TASK(Initiator) {
2    osss::osss_port<
3      osss_shared_if<
4        osss_mediator_initiator_if<AudioCodec_if> > > mediator_so_port;
5
6    OSSS_SW_CTOR(Initiator) : mediator_so_port("mediator_so_port") {}
7
8    protected:
9     // pre-defined main process of software task
10   void main() {
11     Dataword d = 16;
12     // switch to MP3Codec<Buffer>
13     this->switch_to(OSSS_MEDIATOR_CLASS_TYPE(AudioCodec_if, mp3codec));
14     while(d != 0) {
15       if (this->inputBufferFull()) {
16         this->decode();
17         while (! this->outputBufferEmpty() ) {
18           Dataword dout = this->getOutput();
19           // do something with dout ...
20         }
21       }
22       this->putInput( d );
23       --d;
24     }
25     // ...
26     // switch to AACCodec<Buffer>
27     this->switch_to(OSSS_MEDIATOR_CLASS_TYPE(AudioCodec_if, aaccodec));
28     // ...
29     // switch to OggVorbisCodec<Buffer>
30     this->switch_to(OSSS_MEDIATOR_CLASS_TYPE(AudioCodec_if, vodbiscodec));
```

```
31      // ...
32    }
33
34    // ------------------------------------------------------------
35    // helper methods
36    // ------------------------------------------------------------
37
38    bool inputBufferFull() {
39      mediator_so_port->req_inputBufferFull();
40      return mediator_so_port->done_inputBufferFull();
41    }
42
43    void decode() {
44      mediator_so_port->req_decode();
45      mediator_so_port->done_decode();
46    }
47
48    bool outputBufferEmpty() {
49      mediator_so_port->req_outputBufferEmpty();
50      return mediator_so_port->done_outputBufferEmpty();
51    }
52
53    Dataword getOutput() {
54      mediator_so_port->req_getOutput();
55      return mediator_so_port->done_getOutput();
56    }
57
58    void putInput(Dataword d) {
59      mediator_so_port->req_putInput(d);
60      mediator_so_port->done_putInput();
61    }
62
63    void switch_to(osss_mediator_types<AudioCodec_if>::class_t t) {
64      mediator_so_port->osss_req_switch(t);
65      mediator_so_port->osss_done_switch();
66    }
67  };
```

Listing A.12: Definition of the software task

The pre-defined `main()` process (line 10) shows how to call methods in a Mediator Pattern. The example scenario starts with a request for switching the configuration of the Recon Object to `MP3Codec<Buffer>` (line 13). The `switch_to` method (line 63) is a user defined helper method that uses the pre-defined `osss_req_switch` (Request for class switch) and `osss_done_switch` (Class switch done). These methods are called on the local OSSS-Port `mediator_so_port` of the Software Task. The same pattern of helper methods is provided for `inputBufferFull` etc.

In the presented example we do not exploit the decoupling of request and acknowledge for a concurrent software implementation. Consider the possibility to call `req_decode`, perform some local computations (e.g. prepare the next frame for decoding), and call `done_decode` after the local intermediate computation has finished. This pipelined communication scheme can result in a much better resource utilisation and throughput in some applications.

The `OSSS_MEDIATOR_CLASS_TYPE` macros are used to help in finding the right class type for each mediator type. They are only needed for class switching.

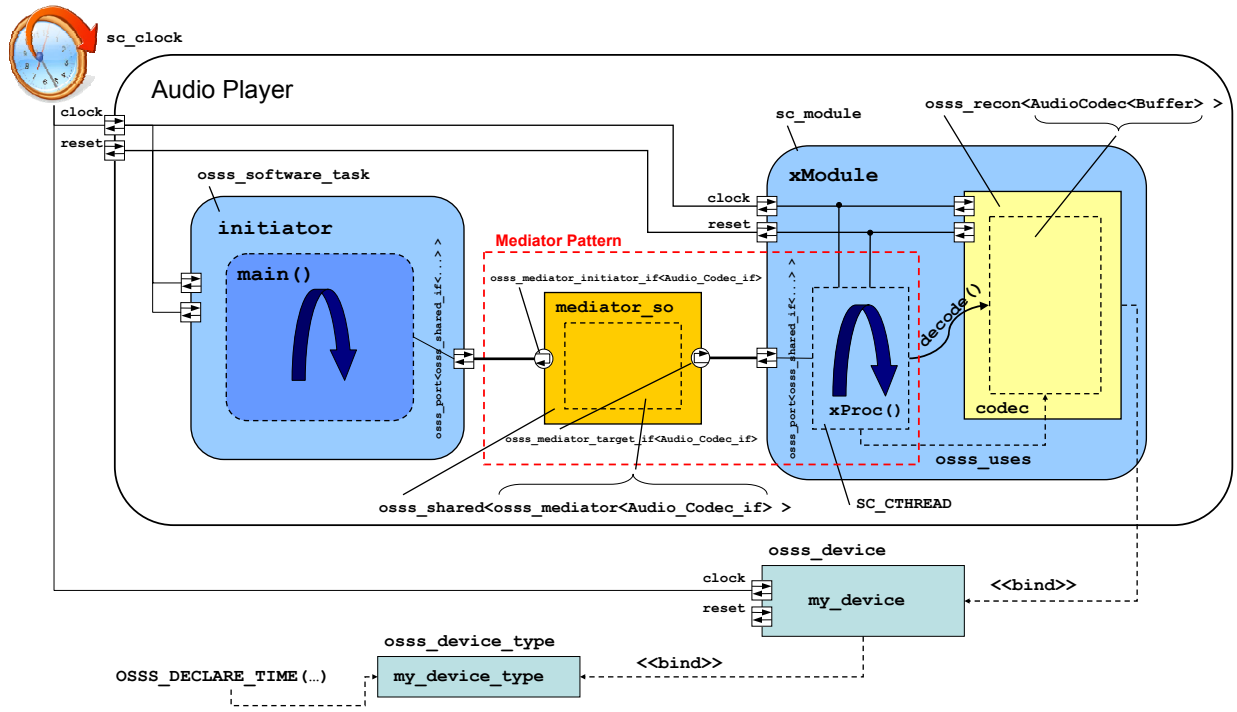



Figure A.9: Structure of reconfigurable audio player using the Mediator Pattern

**Building Application Layer Structure** In this part we are putting all the above Mediator Pattern parts together, ending up with the structure as shown in Figure A.9. The Mediator Pattern itself is marked by a red dotted box.

Listing A.13 defines the Application Layer top level design from Figure A.9. It consists out of the initiator Software Task, the Mediator Shared Object, and the hardware module containing the transactor process.

```
1  SC_MODULE(Top) {
2    sc_in< bool > clock;
3    sc_in< bool > reset;
4
5    Initiator                                          initiator;
6    osss::osss_shared< osss_mediator<AudioCodec_if> > mediator_so;
7    xModule                                            xmodule;
8
9    SC_CTOR(Top) :
10     initiator("initiator"),
11     mediator_so("mediator_so"),
12     xmodule("xmodule")
13   {
14     initiator.clock_port(clock);
15     initiator.reset_port(reset);
16     initiator.mediator_so_port(mediator_so);
17
18     mediator_so.clock_port(clock);
19     mediator_so.reset_port(reset);
20
21     xmodule.clock(clock);
22     xmodule.reset(reset);
```

```
23      xmodule.mediator_so_port(mediator_so);
24    }
25  };
```

Listing A.13: Application Layer Top Level Design using Mediator Pattern

Please note the `mediator_so_port` bindings from the initiator SW Task and the HW module to the `mediator_so` Shared Object. They constitute the communication links that are mapped to physical communication resources during a refinement to the Virtual Target Architecture Layer Model.

### Virtual Target Architecture Layer Model

In this last part of the mediator pattern example we demonstrate how to map the Application Layer Model to a Virtual Target Architecture. It requires three basic steps:

1. `osss_shared_if` → `osss_rmi_if` substitution.
2. OSSS-RMI stub provision.
3. Architecture assembly and Communication Link to OSSS-RMI Channel mapping.

The first step is performed by replacing all `osss_shared_if` interfaces by an `osss_rmi_if` interface. For the second step we need to define initiator and target interface stubs for RMI. These stub definitions are shown in Listing A.14 and Listing A.15.

```
OSSS_MEDIATOR_INITIATOR_STUB_FOR( AudioCodec_if ) {
  OSSS_MEDIATOR_INITIATOR_STUB_CTOR( AudioCodec_if );

  OSSS_MEDIATOR_INITIATOR_STUB_METHOD_VOID( encode, OSSS_VOID );
  OSSS_MEDIATOR_INITIATOR_STUB_METHOD_VOID( decode, OSSS_VOID );
  // ... other methdos from AudioCodec_if ...
};
```

Listing A.14: Definition of initiator stub for RMI

```
OSSS_MEDIATOR_TARGET_STUB_FOR( AudioCodec_if ) {
  OSSS_MEDIATOR_TARGET_STUB_CTOR( AudioCodec_if );

  OSSS_MEDIATOR_TARGET_STUB_METHOD_VOID( encode, OSSS_VOID );
  OSSS_MEDIATOR_TARGET_STUB_METHOD_VOID( decode, OSSS_VOID );
  // ... other methdos from AudioCodec_if ...
};
```

Listing A.15: Definition of target stub for RMI

The architecture assembly and Communication Link to OSSS-RMI Channel mapping is demonstrated in Listing A.16.

```
const unsigned int P2P_SEND_BITSIZE   = 32;
const unsigned int P2P_RETURN_BITSIZE = 32;

class Top : public xilinx::xilinx_system {
public:
  sc_in< bool > clock;
  sc_in< bool > reset;
```

```
8
9   typedef osss_rmi_channel<xilinx::xilinx_opb_channel<> > Bus_Ch_t;
10  typedef osss_rmi_channel<
11    osss_rmi_point_to_point_channel< P2P_SEND_BITSIZE,
12                                     P2P_RETURN_BITSIZE> > P2P_Ch_t;
13
14  xilinx::xilinx_microblaze                              processor;
15  Bus_Ch_t                                               processor_bus;
16  P2P_Ch_t                                               mediator_p2p_ch;
17  Initiator                                              initiator;
18  osss_object_socket<
19    osss_shared< osss_mediator<AudioCodec_if> > >        mediator_so;
20  xModule                                                xmodule;
21
22  Top(sc_core::sc_module_name name) :
23    xilinx::xilinx_system(name),
24    processor("processor"),
25    processor_bus("processor_bus"),
26    mediator_p2p_ch("mediator_p2p_ch"),
27    initiator("initiator"),
28    mediator_so("mediator_so"),
29    xmodule("xmodule")
30  {
31    processor_bus.clock_port(clock);
32    processor_bus.reset_port(reset);
33
34    mediator_p2p_ch.clock_port(clock);
35    mediator_p2p_ch.reset_port(reset);
36
37    initiator.mediator_so_port(mediator_so);
38
39    processor.clock_port(clock);
40    processor.reset_port(reset);
41    processor.rmi_client_port(processor_bus);
42    processor.add_sw_task(initiator);
43
44    mediator_so.clock_port(clock);
45    mediator_so.reset_port(reset);
46    mediator_so.bind(processor_bus);
47    mediator_so.bind(mediator_p2p_ch);
48
49    xmodule.clock_port(clock);
50    xmodule.reset_port(reset);
51    xmodule.mediator_so_port(mediator_p2p_ch, mediator_so);
52  }
53 };
```

Listing A.16: Virtual Target Architecture Top Level Design using Mediator Pattern

OSSS VTA (Virtual Target Architecture) Models are either represented by `osss_system` (implementation platform has not been specified yet) or `xilinx_system` (implementation platform using Xilinx IP Cores). In this example we specify a Xilinx platform (line 4). It consists out of a Xilinx MicroBlaze processor (line 14), a Xilinx derivation of the IBM On-Chip Peripheral Bus (OPB) (line 9 & 15), a user-defined point-to-point channel (line 10-12 & 16) with scalable send and return bit sizes (line 1 & 2), the initiator Software Task

(line 17) from the Application Layer, the Mediator Shared Object (line 18 & 19) from the Application Layer (here it is wrapped by an OSSS Object Socket for RMI connectivity), and the hardware module (line 20) with the transactor process and the Recon Object from the Application Layer.
The top level module constructor (line 22) connects all these architecture blocks with the global clock and reset port, connects the initiator Software Task with the Mediator Shared Object (line 37), connects the Microblaze processor to the OPB (line 41), adds the initiator Software Task to the processor (line 42), binds the Mediator Shared Object to the OPB (line 46) and the user-defined point-to-point channel (line 47), and connects the hardware module's port physically to the point-to-point channel and logically to the Mediator Shared Object (line 51).
A second possible communication link mapping for this example would have been the use of a single OPB. The modified body of the constructor from Listing A.16 is shown in Listing A.17.

```
1  Top(sc_core::sc_module_name name) //...
2  {
3    processor_bus.clock_port(clock);
4    processor_bus.reset_port(reset);
5
6    initiator.mediator_so_port(mediator_so);
7
8    processor.clock_port(clock);
9    processor.reset_port(reset);
10   processor.rmi_client_port(processor_bus);
11   processor.add_sw_task(initiator);
12
13   mediator_so.clock_port(clock);
14   mediator_so.reset_port(reset);
15   mediator_so.bind(processor_bus);
16
17   xmodule.clock_port(clock);
18   xmodule.reset_port(reset);
19   xmodule.mediator_so_port(processor_bus, mediator_so);
20 }
```

Listing A.17: Virtual Target Architecture using single bus only

### A.3.3 Reconfigurable Audio Player with Named Contexts

In this section we use Named Contexts in the reconfigurable audio player example from Section A.3.3. To motivate the use of Contexts we have introduced two processes that both make use of the same reconfigurable area. Figure A.10 presents and updated structure of the previously presented player in Figure A.5.
The motivation for the use of Named Contexts in this example is:

1. each process has its own consistent view of the reconfigurable resource,
2. allows separation of the logical (Context) and the physical (Recon Object) view of dynamic reconfiguration,
3. and allows automatic state preservation and restoration.

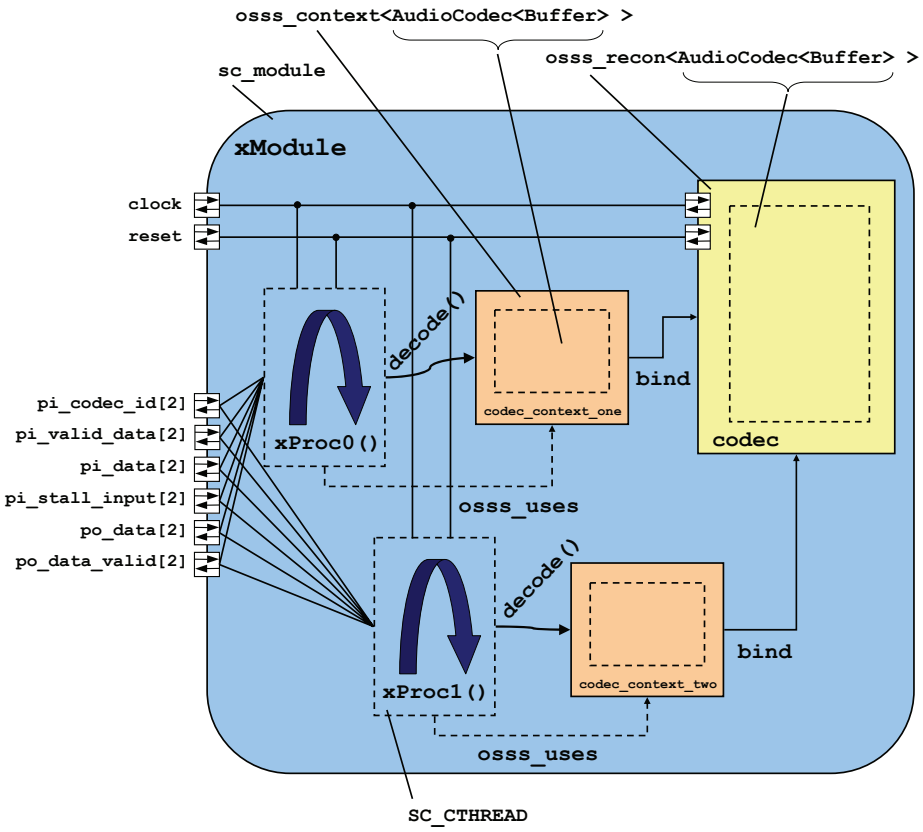



Figure A.10: Structure of reconfigurable audio player with Named Contexts

### Building Structure

Adding Named Context to an OSSS+R design is rather simple. Listing A.3 shows the definition of an audio player module with a single Recon Object (line 8), two Named Contexts (line 11 & 12), and two processes (line 25 & 30).

Process `xProc0` only uses `codec_context_one` (line 35) and process `xProc1` only uses `codec_context_two` (line 40). As we are using a single Recon Object in this example, both Named Contexts need to be bound to the `codec` Recon Object (line 29 & 30).

```
1  SC_MODULE(xModule) {
2    sc_in< bool > clock;
3    sc_in< bool > reset;
4    // ... the other ports ...
5
6    // We want one reconfigurable area in our design.
7    // It shall provide the interface "AudioCodec<Buffer>".
8    osss::osss_recon< AudioCodec<Buffer> > codec;
9
10   // A Named Context with interface "AudioCodec<Buffer>".
11   osss::osss_context< AudioCodec< Buffer > > codec_context_one;
12   osss::osss_context< AudioCodec< Buffer > > codec_context_two;
13
14 protected:
15   void flush(CodecID current_codec_id);
16   void xProc0();
17   void xProc1();
18
19 public:
20   SC_CTOR(xModule) : codec("codec") {
21     // connect Recon Object with clock & reset
22     codec.clock_port(clock);
23     codec.reset_port(reset);
24
25     // Since we use contexts as "virtual reconfigurable areas",
26     // we need to tell the library where they are mapped to.
27     // Here we map the Named Contexts codec_context_one
28     // and codec_context_two to the Recon Object codec.
29     codec_context_one(codec);
30     codec_context_two(codec);
31
32     SC_CTHREAD(xProc0, clock.pos());
33     reset_signal_is(reset, true);
34     // The process xProc0 uses the codec_context_one only
35     osss_uses(codec_context_one);
36
37     SC_CTHREAD(xProc1, clock.pos());
38     reset_signal_is(reset, true);
39     // The process xProc1 uses the codec_context_two only
40     osss_uses(codec_context_two);
41   }
42 };
```

Listing A.18: Definition of audio player module with Named Contexts

### Using Named Contexts

When using Named Contexts all operations (class switch, method call) are performed directly on the Named Context itself. Therefore, all assignments to Recon Objects (class switch) and all applications of `operator->()` (method call) need to be performed on the respective Context. Listing A.19 illustrates these changes in red colour.
This variant of the audio player is a good example for the problem of *context trashing*, as discussed in Section 6.1.8. Both processes `xProc0` and `xProc1` are using the same reconfigurable resource concurrently, which leads to frequent reconfigurations, in case the currently used classes differ. This can e.g. be reduced by adding appropriate `OSSS_KEEP_ENABLED()` or `OSSS_KEEP_PERMISSION()` blocks respectively. Recently, there has been work in combining these techniques with synchronisation via ICODES Shared Objects. The pattern of *Adaptive Locking* has been published in [GON08].

```
1  void xModule::xProc0() {
2    // ...
3
4    while (true) {
5      // ...
6      if (codec_change || no_input) {
7        // ...
8
9        // New codec
10       if (codec_change && (pi_codec_id.read() != NO_ID)) {
11         switch( pi_codec_id.read() ) // switch to next codec
12         { case MP3_ID: codec_context_one = MP3Codec<Buffer>(); break;
13           case AAC_ID: codec_context_one = AACCodec<Buffer>(); break;
14           case OGG_ID: codec_context_one = OggVorbisCodec<Buffer>();
15             break;
16         }
17       }
18       // ...
19     } else { // no codec change and valid input
20       if ( ! codec_context_one->inputBufferFull() ) {
21         //...
22         codec_context_one->putInput( d );
23       } else {
24           wait();
25       }
26     }
27   } // while
28 }
29
30 void xModule::xProc1() {
31   // ...
32   while (true) {
33     // ...
34
35       if (codec_change && (pi_codec_id.read() != NO_ID)) {
36         switch( pi_codec_id.read() ) // switch to next codec
37         { case MP3_ID: codec_context_two = MP3Codec<Buffer>(); break;
38           case AAC_ID: codec_context_two = AACCodec<Buffer>(); break;
39           case OGG_ID: codec_context_two = OggVorbisCodec<Buffer>();
40             break;
41         }
```

```
42          }
43        // ...
44      } // while
45  }
```

Listing A.19: Definition of audio player module process using Named Contexts

The rest of the reconfigurable audio player design is directly not affected by the introduction of Named Contexts. Of course, the testbench needs to be adapted for driving both processes `xProc0` and `xProc1` independently.

### A.3.4 Reconfigurable Audio Player with Prefetching

To enable prefetching in the previously described audio player design, the initial "plain" Recon Object variant is extended. It is assumed, that the `pi_codec_id` port is set by the testbench early enough to allow the (potentially) required reconfiguration to complete, before the newly added triggering signal `pi_switch_codec` is asserted.

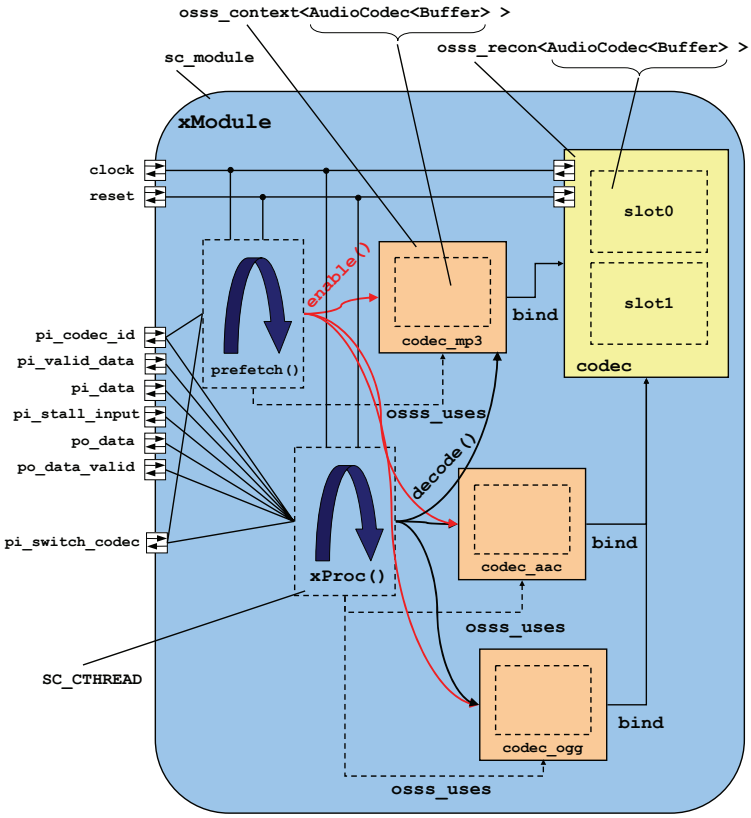



Figure A.11: Structure of reconfigurable audio player with configuration prefetching

In addition to enabling the multi-slot feature (see Section 7.2.2) of the contained Recon Object, a new configuration prefetching process `prefetch` is added, as well as three specific contexts `codec_mp3`, `codec_aac`, and `codec_ogg`. This modelling style simplifies the modelling of the processes, since no explicit communication, which context to use, is needed. In Listing A.20 the changes to the original audio player module are highlighted. The resulting structure of the player model is shown in Figure A.11.

```
SC_MODULE(xModule) {
  sc_in< bool > clock;
  sc_in< bool > reset;

  sc_in< int >        pi_codec_id;      // next codec to fetch
  sc_in< bool >       pi_switch_codec; // trigger switch input
  // ... other ports

  osss_recon< AudioCodec<Buffer> >    codec;
  // contexts for prefetching
  osss_context<AudioCodec<Buffer > > codec_aac;
  osss_context<AudioCodec<Buffer > > codec_mp3;
  osss_context<AudioCodec<Buffer > > codec_ogg;

protected:
  void flush(CodecID);
  void xProc();
  void prefetch(); // prefetching control
public:
  SC_CTOR(xModule) : codec("codec") {
    // ... bindings of osss_recon and osss_context's

    // multi-slot recon object
    codec.setNumSlots( 2 );
    codec.setNumParallelAccessesAllowed( 2 );

    SC_CTHREAD(prefetch, clock.pos()); // prefetching process
    reset_signal_is(reset, true);
    osss_uses(codec);

    SC_CTHREAD(xProc, clock.pos());     // computation process
    // ...
  }
};
```

Listing A.20: Definition of Audio Player Module with Configuration Prefetching

The codec ID is then read by the control process `prefetch`, which then `enable()`s the appropriate context. Since the request of different contexts is assumed to be reasonably seldom, no explicit synchronisation is modelled within the prefetching process. It only waits for the `pi_switch_codec` input port to be asserted, before looking for changes in the requested codec again.

```
void xModule::prefetch() {
  CodecID current_codec_id = NO_ID;
  // ... context initialisiation omitted
  while( true ) {
    // new codec to be fetched?
    bool codec_change = (pi_codec_id.read() != current_codec_id);
    if( codec_change ) {
      current_codec_id = pi_codec_id.read();
      switch( current_codec_id ) {
        case MP3_ID :
          codec_mp3.enable(); break; // enable MP3
        case AAC_ID :
          codec_aac.enable(); break; // anable AAC
```

```
14          case OGG_ID :
15            codec_ogg.enable(); break; // enable OggVorbis
16        }
17        // wait for signal to switch
18        while( pi_switch_codec.read() == false ) wait();
19      }
20    } // main loop
21  }
```

Listing A.21: Definition of the Configuration Prefetching Process

The computation process `xProc` can then simply use always the currently active codec, and updates this in case the switch is triggered. The main difference in the implementation is the use of three contexts within the same process. This removes the necessity of explicit inter-process communication between the prefetching process and the computation process about which "slot" i.e. context should be used at a given time. On the other hand, each call has to be replaced by a corresponding `switch` statement, as shown in Listing A.22.
To enable the prefetching behaviour, the most important change is the addition of the `OSSS_KEEP_ENABLED()` block around the main process loop. This has already been discussed in Section 7.2 and is required to decouple the reconfiguration requests issued by the prefetching controller from the slot accesses, that are requested concurrently by the computation process.

```
22
23  void xModule::flush() {
24    switch( current_codec_id ) {
25      // simply use currently requested codec
26      case MP3_ID:
27        codec_mp3->decode();
28        while( ! codec_mp3->outputBufferEmpty() ) {
29          po_data.write( codec_mp3->getOutput() );
30          po_data_valid.write( true );
31          wait();
32          po_data_valid.write( false );
33        }
34        break;
35      case AAC_ID:
36        codec_aac->decode();
37        // ...
38        break;
39      case OGG_ID:
40        // ...
41        break;
42    }
43  }
44
45  void xModule::xProc() {
46    CodecID current_codec_id = NO_ID;
47    // ... reset
48
49    OSSS_KEEP_PERMISSION( codec ) // keep permission on recon object
50      while (true) {
51        po_stall_input.write( true );
52
53        // is input available?
```

```
54        bool no_input = (pi_data_valid.read() == false);
55        if ( no_input ) {
56          flush(current_codec_id);
57        }
58
59        // switch to (hopefully) prefetched codec
60        if( pi_switch_codec.read() == true ) {
61          current_codec_id = pi_switch_codec.read();
62        }
63
64        // process input, if available
65        if( ! no_input ) {
66          // ...
67        } else {
68          wait(); // do nothing
69        }
70    } // while && OSSS_KEEP_PERMISSION
71  }
```

Listing A.22: Definition of Computation process of Audio Player with Configuration Prefetching

## A.4 SystemC examples

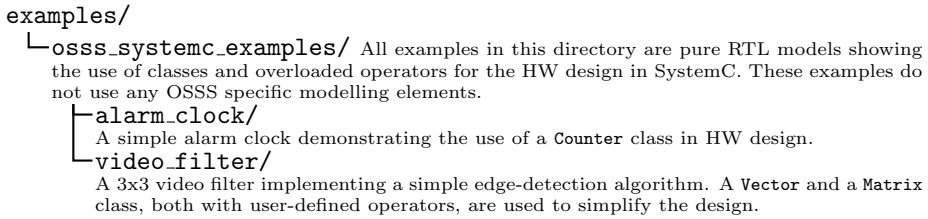



Figure A.12: Structure of the OSSS 2.2.0 simulation library `osss_systemc_examples` directory

# Index

# References and Further Reading


[A. 05] A. Rose, S. Swan, J. Pierce and J.-M. Fernandez. Transaction Level Modeling in SystemC. Whitepaper, OSCI TLM Working Group, 2005.

[agi] Agility. www.agilityds.com. 24

[AND] ANDRES *project*. http://andres.offis.de. 11

[cat] CatapultC. www.mentor.com. 24

[cpp] C++ Portability Guide, Version 0.8. http://www.mozilla.org/hacking/portable-cpp.html. 70

[cpp98] *ISO/IEC 14882 C++ Standard*, first edition, September 1998. 29, 132, 138

[cyn] Cynthesizer. www.forteds.com. 24

[DDM+07] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A Next-Generation Design Framework for Platform-Based Design. In *Conference on Using Hardware Design and Verification Languages (DVCon)*, February 2007. 24

[edg] Homepage of Edison Design Group (EDG). http://www.edg.com. 138

[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. 25, 66

[gcca] GNU Extensions to the C Language Family. http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html.

[gccb] Homepage of the GNU Compiler Collection (gcc) Project. http://gcc.gnu.org/. 17, 132

[GGON07] Kim Grüttner, Cornelia Grabbe, Frank Oppenheimer, and Wolfgang Nebel. Object Oriented Design and Synthesis of Communication in Hardware-/Software Systems with OSSS. In *Proceedings of the SASIMI 2007*, October 2007.

[GGS05] Cornelia Grabbe, Kim Gruettner, and Thorsten Schubert. Specification of Hardware/Software Communication Design Methodology based on Abstract Communication Models. ICODES deliverable D11, OFFIS Institute for Information Technology, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. 105

[GLMS02] Thorsten Groetker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002. 18

[GO03] Eike Grimpe and Frank Oppenheimer. Extending the SystemC Synthesis Subset by Object Oriented Features. In *CODES + ISSS 2003*, October 2003.

[GON08] Ralph Görgen, Frank Oppenheimer, and Wolfgang Nebel. Adaptive Scheduling of Dynamic Reconfiguration in Stream-Based Applications. In *Proceedings of the ReCoSoC'08*, Barcelona, Spain, July 2008. 63, 183

[gre] Homepage of Greensocs. http://www.greensocs.com.

[Gro04] OSCI Synthesis Working Group. SystemC Synthesisable Subset. Technical Report Draft 1.1.18, Open SystemC Initiative, 23 December 2004. 31

[GTF+02] Eike Grimpe, Bernd Timmermann, Tiemo Fandrey, Ramon Biniasch, and Frank Oppenheimer. SystemC Object-Oriented Extensions and Synthesis Features. In *Forum on Design Languages FDL'02*, September 2002.

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Springer, 2000. 24

[has] Haskell general purpose, purely functional programming language homepage. http://www.haskell.org. 138

[IBM] IBM. *On-Chip Peripheral Bus Architecture Specifications, Version 2.1, SA-14-2528-02*. 67, 84, 91

[ICOa] ICODES *project*. http://icodes.offis.de. 11

[icob] Homepage of the ICODES Project. http://icodes.offis.de.

[IEE05] IEEE Standards Association ("IEEE-SA") Standards Board. *IEEE Std 1666-2005 Open SystemC Language Reference Manual*, 2005. 20

[IEE06] IEEE. *IEEE 1666™ Open SystemC Language Reference Manual*, 2006. 11, 13, 24

[ISO98] ISO/IEC. *ISO/IEC 14882:1998(E) Programming Languages - C++*, First Edition 1998.

[J. 06] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-Based Behavior-Level and System-Level Synthesis. In *IEEE International SOC Conference*, September 2006. 24

[KG05] Wolfgang Klingauf and Robert Günzel. From TLM to FPGA: Rapid Prototyping with SystemC and Transaction Level Modeling. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, December 2005. 24

[LCH07] Zhiyuan Li, Katherine Compton, and Scott Hauck. Configuration Caching Management Techniques for Reconfigurable Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (submitted)*, 2007. 111

[LH02] Zhiyuan Li and Scott Hauck. Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation. In *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pages 187–195, 2002. 111

[lib] An update to AGM::LibReflection: A reflection library for C++. http://www.codeproject.com/library/libreflection2.asp.

[mbg] Homepage of the GNU Compiler Collection for the Xilinx MicroBlaze Soft Processor Core (mb-gcc). http://www.xilinx.com/guest_resources/gnu/index.htm. 132

[ODEa] ODETTE *project*. http://odette.offis.de. 11

[odeb] Homepage of the ODETTE Project. http://odette.offis.de.

[OSC] Open SystemC Initiative. *OSCI TLM2.0*. http://www.systemc.org. 12, 24

[Pol] PolyDyn *project*. http://ehs.informatik.uni-oldenburg.de/en/research/projects/polydyn. 13

[RSPF05] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction Level Modeling in SystemC. Technical report, Mentor Graphics; Cadence Design Systems, 2005. 12, 31

[SON06] Andreas Schallenberg, Frank Oppenheimer, and Wolfgang Nebel. OSSS+R: Modelling and Simulating Self-Reconfigurable Systems. In *Proceedings - 2006 International Conference on Field Programmable Logic and Applications*, pages 177–182, August 2006. 14

[sou] Homepage of sourceforge. http://sourceforge.net/projects/greensocs/.

[SV07] Alberto Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007. 23

[syn] Homepage of Synplicity. http://www.synplicity.com/. 118

[Syn04] Synthesis Working Group Members of Open SystemC Initiative. SystemC Synthesizable Subset, Draft 1.1.18. Whitepaper, Open SystemC Initiative (OSCI), December 2004. 27, 28

[sysa] Homepage of the Open SystemC Initiative. http://www.systemc.org. 17, 18

[sysb] Homepage of the OSSS and *Fossy* Project. http://www.system-synthesis.org. 21, 29

[sys06] *IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2005*, March 2006. 17, 117

[TCJW97] S. Trimberger, D. Carberry, A. Johnsons, and J. Wong. A Time-Multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, 1997. 111

[VIR] Xilinx Inc. *Virtex$^{TM}$-4 Multi-Platform FPGA*. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/. 13

[xila] Xilinx website. http://www.xilinx.com. 23

[xilb] Xilinx Platform Studio and the Embedded Development Kit (EDK) Documentation Homepage. http://www.xilinx.com/ise/embedded/edk_docs.htm. 117

[xilc] Xilinx ISE Software Manuals Homepage. http://www.xilinx.com/support/software_manuals.htm. 117

[Xil05] Xilinx. *On-Chip Peripheral Bus V2.0 with OPB Arbiter (v1.10c), DS401*, June 2005. 67, 84, 91