OSSS+R
Getting Started

A. Herrholz
July 2, 2008
andreas.herrholz@offis.de
Rev. 1.1

Contents

1 Introduction **3**

2 Motivation and Scope **3**

3 Overview **3**
3.1 General Concept 3

4 Using OSSS+R **6**
4.1 Installation 6
4.1.1 Version Information 6
4.1.2 System Requirements 6
4.1.3 Installation Instructions 6
4.2 Example: *audio* 7
4.2.1 Defining Classes 7
4.2.2 Implementing Methods 8
4.2.3 Building Structure 8
4.2.4 Using the Recon-Object 10
4.2.5 Creating a Testbench 10
4.2.6 Running the Example 11
4.2.7 Using contexts: *two_process_audio* 11
4.3 What's next? 12

5 Contact **13**

1 Introduction

The following document has been created as part of the ANDRES project. It is a short introduction to OSSS+R. It presents the concept and some of the basic modelling elements. The language is explained using a simple example of a reconfigurable audio player. Note, that this is not a complete user's guide or a language reference manual. However it should give a first impression of OSSS+R and how it is used to model reconfigurable hardware systems.

2 Motivation and Scope

The starting point in ANDRES for building a modelling framework for adaptive and heterogeneous systems are three different SystemC extensions: SystemC-AMS, HetSC and OSSS+R. Understanding the principles and target domains of each extension is essential for further collaboration and successful work on the ANDRES modelling framework. Therefore this short introduction is intended for those partners within ANDRES that want to get to know OSSS+R and understand its basic concepts without further investigation of its source code.

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

Please note that OSSS+R is currently under ongoing development and some of the concepts and language elements may change in future.

3 Overview

OSSS+R is a SystemC based software library for high-level modelling of reconfigurable digital hardware systems. Although OSSS+R shares some of its concepts with OSSS (Oldenburg System Synthesis Subset) it can be used stand-alone. However also a combination of both is possible.

OSSS+R has been designed with synthesis in mind. That means all modelling elements that can be used in a OSSS+R design have a well-defined synthesis semantic. A synthesis tool that integrates the synthesis concepts for OSSS+R will be developed within ANDRES.

The next section describes the general concept of the OSSS+R methodology without giving any details of the language. To see how these concepts are expressed in the OSSS+R language see the examples in section 4.2.

3.1 General Concept

In OSSS+R objects are used to represent possible configurations of a reconfigurable area (Fig. 1). These objects contain methods and attributes and are defined by their underlying class. Every object has a state which is given by the current values of its attributes. If an object becomes inactive, i.e. it is removed from the reconfigurable area, its state is saved, so it can be restored if the object is reactivated.

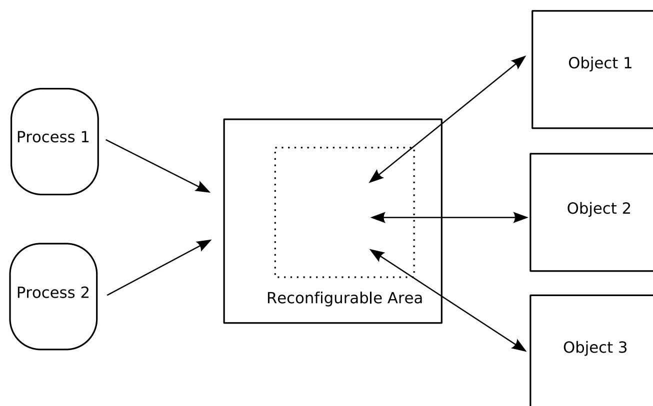

Figure 1: Using objects as configurations of a reconfigurable area

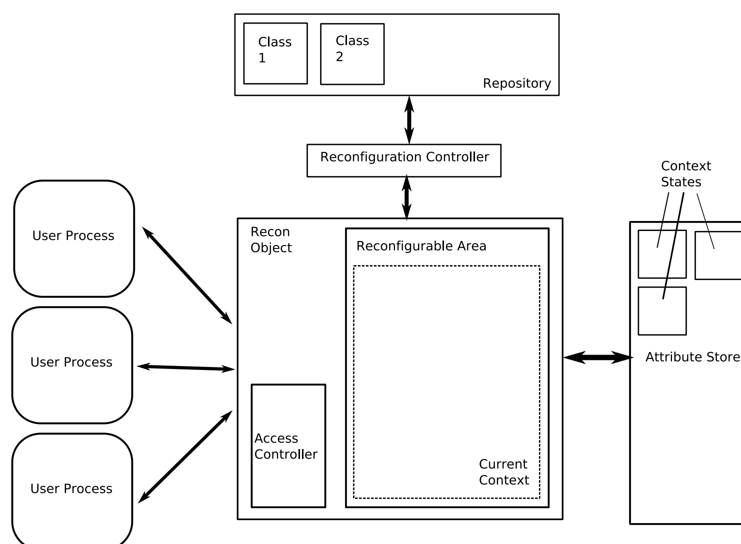

Figure 2: OSSS+R architecture

OSSS+R imitates the concept of software **contexts** to keep track of the available object configurations of a reconfigurable area. In OSSS+R every context represents one possible configuration, which is an object including its current state. There is usually more than one context available for a reconfigurable area, but there can only be one active context at a time. The state of a context is saved when it becomes inactive and restored when it becomes active.

From the user's point of view contexts are similar to pointers in C++. Every context identifies one object, so all objects that are to be used with a reconfigurable area need to be assigned to a context and can only be accessed using contexts. If a new object is assigned to a context, its old state is lost.

Every context that is used in an OSSS+R design must be bound to a reconfigurable area. These areas are called **Recon-Objects** (Fig. 2) and provide all infrastructure for context access, context storage and context switches. Because a Recon-Object can be used by more than one process, it needs to have a built-in access controller that serialises all incoming requests and decides when to switch contexts. It is also connected to the **attribute storage**, a memory area that is used to store the state of the context during a context switch.

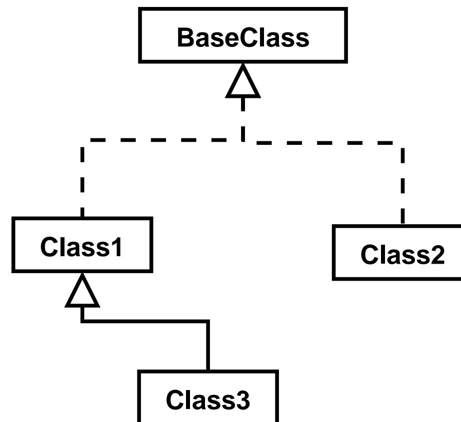

Figure 3: Sample class hierarchy

A **context switch** occurs if a user process requests access to a context that is currently not available in the Recon-Object. There are two types of context switches

- The context classes are equal.
- The context classes are different.

In both cases the current context's state is saved to the attribute storage and the state of the other context is restored. A reconfiguration, i.e. the change of hardware circuits, does only need to take place if the two classes are different.

Although the contexts of one Recon-Object may belong to different classes, all classes need to be part of the same class hierarchy, i.e. they need to inherit from the same base class (see Fig. 3). This base class is part of the definition of a Recon-Object, so only classes derived from the base class can be used with the corresponding recon object.

Recon-objects and their contexts can be accessed by user processes. These user processes are either hardware processes that reside in fixed areas or software processes running on a connected processor. A user process has to register itself to the recon object before it can define and use contexts. Hardware processes in OSSS are usually SystemC *CThreads* representing implicit state machines. Communication between user processes and contexts is specified using method calls. All methods are executed within the thread of the running process, i.e. they are blocking and are allowed to include wait-statements. [1]

For some use cases it may not be needed to have the feature of a context switch, e.g. if there is only one user process or the objects' methods are independent of their states. In this case objects can directly be assigned to a Recon-Object. This results in the creation of a **temporary context**. This temporary context is available as long as no other object is assigned to a Recon-Object. After the temporary context has been destroyed it can not be restored.

[1] This may be extended to non-blocking calls in future.

4 Using OSSS+R

4.1 Installation

4.1.1 Version Information

The current version of the OSSS+R library is the initial version for Deliverable 1.3a. It contains at least the basic functions that are described within this document. Note that some of the concepts and elements described herein might change or extended with a later release. However most of the basic concepts will stay the same.

4.1.2 System Requirements

Currently the minimal requirements for successful compilation of the OSSS+R simulation library and OSSS+R designs is a GNU/Linux based system with installed SystemC 2.1, GCC 3.4.2 and MAKE 3.79. OSSS+R has also successfully been tested with SystemC 2.2 and GCC 4.1.X. Other platforms may also work, but have not been tested to date.

When building the OSSS+R simulation library, the provided configuration file is trying to guess the location of your SystemC installation. However you can explicitly give the location by setting the environment variables *SYSTEMC_HOME* and *SYSTEMC_LIB* (The first one points to the installation path and the second one to the location of your SystemC library). Possible examples are:

- If your log-in shell is *bourne shell-like* (bash, sh, ...):
  `export SYSTEMC_HOME=/home/user/systemc/systemc-2.2/`
  `export SYSTEMC_LIB=${SYSTEMC_HOME}/lib-linux-gcc`
- If your log-in shell is *c-shell-like* (csh, tcsh, ...):
  `setenv SYSTEMC_HOME /home/user/systemc/systemc-2.2/`
  `setenv SYSTEMC_LIB ${SYSTEMC_HOME}/lib-linux`

4.1.3 Installation Instructions

1. Download the OSSS+R installation package from the ANDRES website.
2. Execute the installation script by entering:
   `./osss_recon-<version>.sh`
3. To extract the library, accept the displayed license.[2].
4. To build OSSS+R, change into the extracted directory and run `make`.
5. If you would like to build the examples run `make examples`.
6. If you encounter any errors during the build process or to see more build options run `make help`.

[2] You may have to press Q to quit the license display.

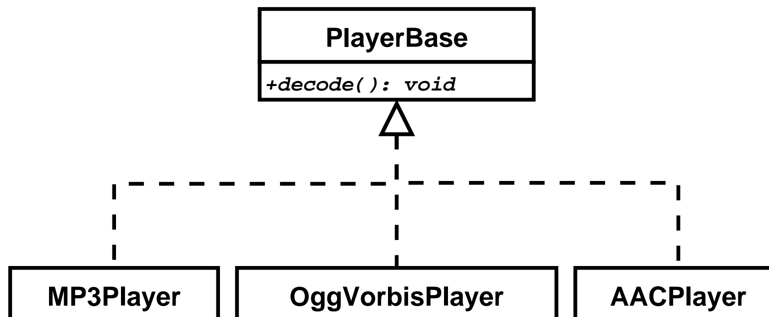

Figure 4: Class hierarchy for audio example

4.2 Example: *audio*

In this example we are designing a simple reconfigurable audio player that is capable of decoding MP3, OggVorbis and AAC data. Note, that some of the code presented in this document has been modified or shortened for readability and/or simplicity. Also, you will find some more specific and detailed comments in the complete source code which can be found in the examples directory of the OSSS+R installation.

The starting point for the design is the given class hierarchy in Fig. 4. There is an abstract base class *PlayerBase* that defines the method *decode()* and three derived classes where each implements *decode()* according to their designated codec.

4.2.1 Defining Classes

Listing 1 shows the definition of the corresponding player classes in OSSS+R. [3]

Listing 1: Definition of player classes

```
1  template<class MemClass>
2  class PlayerBase : public osss::osss_object
3  {
4   public:
5
6     PlayerBase()
7     {
8       OSSS_BASE_CLASS( osss::osss_object );
9     }
10
11    virtual void decode(MemClass & /*compressed*/,
12                        MemClass & /*decompressed*/){};
13 };
14
15 template<class MemClass>
16 class MP3Player : public PlayerBase<MemClass>
17 {
18  public:
19
20    MP3Player()
21    {
22      OSSS_BASE_CLASS(PlayerBase<MemClass>);
24
```

[3]The template parameter *MemClass* may be any class that implements the methods *read(int addr)* and *write(int addr)*

```
23      }
25
26      virtual void decode(MemClass & compressed,
27                          MemClass & decompressed);
28
29  };
```

Classes are described in OSSS+R like in C++ except for some additional elements: Every class that is used with a Recon-Object needs to be derived from *osss_object*. In this example the base class *PlayerBase* is explicitly derived from *osss_object*. All other classes are derived from *PlayerBase*. Additionally every class needs to use the macro *OSSS_BASE_CLASS* in its constructor to indicate its direct base class. [4]

Note that classes that are to be used within recon objects cannot have constructors with parameters. If you need to initialise class member you have to define an init-method and call it directly after object construction.

4.2.2 Implementing Methods

Methods may be declared, implemented and overloaded within any class of the given hierarchy. As in standard C++, to use polymorphism, the methods need to be declared as being virtual in the common base class. When synthesising the design, the resulting interface of the Recon-Object will include the set of all methods of all classes that can potentially be used with the Recon-Object.

As previously said, all methods are executed within the context of the calling user process. That is, they are blocking and may contain wait-statements [5]. In our example we have only one method *decode()* which is implemented in every class. The implementation of *decode()* in *MP3Player* is given in Listing 2. In this example the method does not implement the decoding algorithm but shows a debug message and does some data manipulation.

Listing 2: Implementation of MP3Player::decode()

```
1   virtual void MP3Player::decode(MemClass & compressed,
2                                  MemClass & decompressed)
3   {
4     OSSS_MESSAGE(true, "MP3 decode");
5     for (int addr = 0; addr < compressed.memsize(); ++addr)
6     {
7       decompressed.write(addr, compressed.read(addr) + 1);
8       wait();
9     }
10  }
```

4.2.3 Building Structure

After the classes have been defined we can create the architecture of the player. In OSSS+R, structure is described using standard SystemC elements like SC_MODULE,

[4]May be removed in future.

[5]Internal functions called by those methods may also contain wait-statements.

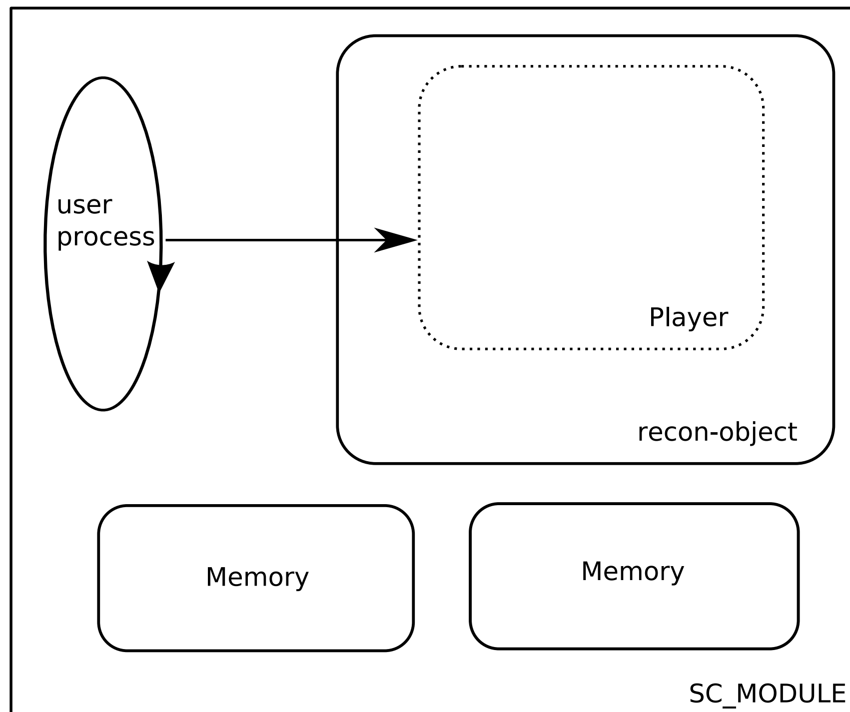

Figure 5: Structural architecture of player

ports and signals. For this example we create DUT, a module that contains one process, one Recon-Object, two memory objects (see Fig. 5) and clock and reset ports.

An excerpt of the definition of DUT is given in Listing 3.

Listing 3: Definition of DUT

```
SC_MODULE(DUT)
{
   ...
   osss::osss_recon< PlayerBase<Memory> > decoder;
   ...
   SC_CTOR(DUT) : decoder("decoder_module")
   {
      decoder.clock_port(clock);
      decoder.reset_port(reset);

      SC_CTHREAD(core, clock);
      reset_signal_is(reset, true);
      osss_uses(decoder);
   }
};
```

Line 4 defines the Recon-Object *decoder* by using the template class *osss_recon* with *PlayerBase<Memory>* as the template parameter. This parameter declares the base class of all objects that are allowed to be used with *decoder*.

The following lines define the constructor of *DUT*. Because every Recon-Object needs a clock and a reset signal the corresponding ports of *DUT* are bound to *decoder*.[6] The method *core()* is declared to be a *SC_CTHREAD*. In OSSS+R, every process that is

[6] The Recon-Object must be connected to the same clock and reset as the user processes

accessing a Recon-Object must be registererd to it using the *osss_uses*-statement. The statement is put directly after the declaration of the corresponding process similar to the *reset_signal_is*- or the *sensitive*-statement.

4.2.4 Using the Recon-Object

Listing 4 shows parts of the definition of the user process *core()*. This example does not use contexts, but the objects are assigned directly to the Recon-Object. Every assignment of a newly created object results in the creation of a temporary context within the Recon-Object. The process consecutively assigns a *MP3Player*, an *OggVorbisPlayer* and an *AACPlayer* object to *decoder*. After each assignment it calls the *decode()* method which is automatically delegated to the current temporary context. Note that the syntax of the method call always uses *decoder* as the target identifier of the call. Every call that uses the Recon-Object as the target is automatically applied to the last assigned object.

Listing 4: Definition of *core()*

```
1  void core()
2  {
3    while (true)
4    {
5      decoder = MP3Player<Memory>();
6      osss_call(decoder)->decode(input_buffer, output_buffer);
7
8      decoder = OggVorbisPlayer<Memory>();
9      osss_call(decoder)->decode(input_buffer, output_buffer);
10
11     decoder = AACPlayer<Memory>();
12     osss_call(decoder)->decode(input_buffer, output_buffer);
13
14     osss_call(decoder)->decode(input_buffer, output_buffer);
15   };
```

4.2.5 Creating a Testbench

To simulate the design we need to build a testbench. The testbench components are instantiated in the *sc_main*-function.

Every design that uses one or more Recon-Objects needs to define at least one device object. The device object represents a FPGA device including its available resources and timing informations. Every Recon-Object must be bound to one device.

Listing 5 shows one possible definition of a OSSS+R test scenario. Line 4 defines the FPGA device type *my_device_type*. The following lines assign reconfiguration time to the combination of device type and configuration object. Line 11 instantiates one device object *my_device* of the defined device type which is then bound to the Recon-Object *decoder*. The last lines show the binding of clock and reset ports of *my_device* to corresponding signals.

Listing 5: Definition of *sc_main*

```
1  int sc_main(int /*argc*/, char * /*argv*/[])
2  {
```

```
3    ...
4    osss::osss_device_type my_device_type("My FPGA device type");
5
6    OSSS_DECLARE_TIME(my_device_type, PlayerBase<Memory>,
7                      sc_time(26, SC_US), sc_time(200, SC_US));
8    OSSS_DECLARE_TIME(my_device_type, MP3Player<Memory>,
9                      sc_time(26, SC_US), sc_time(200, SC_US));
10   ...
11   osss::osss_device my_device(my_device_type, "my_device");
12
13   dut.decoder( my_device );
14   my_device.clock_port(clock);
15   my_device.reset_port(reset);
16   ...
17 }
```

4.2.6 Running the Example

The example can be compiled and simulated like standard SystemC designs. However you must make sure, that you are adding the include directory of OSSS+R and linking to the OSSS+R simulation library. For own designs, you may also edit the supplied Makefiles of the examples to your needs.

4.2.7 Using contexts: *two_process_audio*

There is one problem with the simple audio example: If there is more than one process that is accessing the Recon-Object using temporary contexts, one process cannot make sure that the context has not been changed between the assignment of an object and the method-call, because another process might have done an assignment between the two actions. That is one of the motivations to use *named* contexts.

The example *two_process_audio* is nearly similar to audio but defines two processes instead of one and uses contexts. Listing 6 shows parts of the definition of *DUT*.

Listing 6: Definition of *DUT* in *two_process_audio*

```
1  SC_MODULE(DUT)
2  {
3    osss::osss_recon< PlayerBase<Memory> > decoder;
4    osss::osss_context< PlayerBase< Memory > > view_of_process_one;
5    osss::osss_context< PlayerBase< Memory > > view_of_process_two;
6
7    void process_one()
8    {
9      OSSS_MESSAGE(true, "process one (re-)started");
10     while (true)
11     {
12       // This process uses MP3 and AAC
13       OSSS_MESSAGE(true, "Switching to MP3");
14       view_of_process_one = MP3Player<Memory>();
15
16       OSSS_MESSAGE(true, "Decoding MP3");
17       osss_call(view_of_process_one)->decode(input_buffer,
             output_buffer);
```

```
18
19        OSSS_MESSAGE(true, "Switching to AAC");
20        view_of_process_one = AACPlayer<Memory>();
21
22        OSSS_MESSAGE(true, "Decoding AAC");
23        osss_call(view_of_process_one)->decode(input_buffer,
              output_buffer);
24      };
25    }
26
27    ...
28
29    SC_CTOR(DUT) : decoder("decoder_module")
30    {
31      decoder.clock_port(clock);
32      decoder.reset_port(reset);
33
34      SC_CTHREAD(process_one, clock);
35      reset_signal_is(reset, true);
36      osss_uses(view_of_process_one);
37
38      SC_CTHREAD(process_two, clock);
39      reset_signal_is(reset, true);
40      osss_uses(view_of_process_two);
41
42      view_of_process_one(decoder);
43      view_of_process_two(decoder);
44    }};
```

Line 4 and 5 define two contexts having the same base class parameter as *decoder*: *view_process_one* and *view_process_two*. In the constructor both of the two processes register to one of the contexts with the *osss_uses*-statement. Additionally the contexts need to be bound to the Recon-Object, which is done using the syntax shown in line 42 and 43.

The implementation of the user processes and the usage of the contexts is quite similar to the audio example. Objects are created and assigned to the contexts instead of the Recon-Object. The contexts are also used as targets for the method calls. Because both processes use different contexts and the context is used as an identifier for the call, it is made sure that the call is always passed to the correct object.

4.3 What's next?

There is still a problem with the use of the contexts in *two_process_audio*: While it is assured that every process addresses the correct object, it might result into repeated context switches causing trashing and a bad ratio of computation to reconfiguration time. This is because each of the other processes might cause a context switch between the assignment and the method call of the other process resulting in two unnecessary context switches.

OSSS+R provides additional features to avoid context thrashing. Some of them are demonstrated in *no_trashing_audio*. Please consult the comments in the source code for more details.

5 Contact

If you have any questions or problems regarding OSSS+R please contact OFFIS at *osss-devel@offis.de*.