OFFIS
INSTITUTE FOR INFORMATION TECHNOLOGY

R&D Division Transportation
Hardware/Software Design Methodology Group

fossy

Predictable SystemC™/C++ Synthesis

The Fossy Manual

Oldenburg 2009

Last compiled: July 7, 2009

Contents

1 Introduction

The following document serves as a quick introduction to the usage of *Fossy*. To understand this manual you should already be familiar with basic concepts of hardware design. Moreover, basic knowledge of SystemC is required since *Fossy* is a compiler that transforms SystemC code into synthesisable VHDL.

1.1 Overview

- *Fossy* stands for **F**unctional **O**ldenburg **S**ystem **SY**nthesiser.
- *Fossy* is a tool for transforming system-level SystemC models to synthesisable VHDL.
- *Fossy* enables a seamless design flow for embedded HW/SW systems.

Fossy has been developed at the OFFIS Institute for Information Technology, an application oriented research and development institute working on system-level design methodologies for more than ten years. Two consecutive research projects (funded by the European Union) in cooperation with leading industrial partners resulted in robust tools with a focus on practical applicability.

This manual is structured as follows:

- The follwing part of Chapter 1 presents the typographical conventions used throughout this manual.
- Chapter 2 gives an overview of the *Fossy* command line tool.
- Chapter 3 presents the SystemC synthesis capabilities of *Fossy*. It start with an general introduction followed by the definition of the supported synthesis subset.
- Chapter 4 provides contact information in case of technical questions and recommands further readings.

1.2 Typographical conventions

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

Convention	**Item**	**Example**
Times New Roman	Normal Text	This is an example sentence.
Times New Roman	Emphasised Text	This *word* is emphasised.
Monospace font	Class, function, method or macro names	execute_operation()
Monospace italics	Variables meant to be replaced when the language construct is used. Sometimes variables or parameters are omitted.	my_class<*parameter*> my_class<...> do_something(...)
Monospace font	Shell commands	make

Table 1.1: Typographical conventions for continuous text

```
#include <systemc.h>

SC_MODULE( my_module ) {
  /* ----- input ports ----- */
  sc_in<bool> clock;
  sc_in<bool> reset;
  // add your input ports here
  // sc_in< sc_uint<32> > in_data;

  /* ----- output ports ----- */
  // add your output ports here
  // sc_out< sc_uint<32> > out_data;

  /* ----- constructor ----- */
  SC_CTOR( my_module ) {
    /* ----- process definitions ----- */
    SC_CTHREAD( processing, clock.pos() );
    reset_signal_is( reset, true );
  }

private:
  /* ----- process(es) ----- */
  void processing();

  /* ----- sub-module(s), and other members ----- */
  // ...
}; // my_module

/* ----- process body ----- */
void
my_module::processing() {
  /* ----- reset block ----- */

  wait();
  while( true ) {
    /* ----- main loop ----- */
    wait();
  }
} // my_module::processing()

/* ----- sc_main() -----
 *
 * This is optional and only required, if you have more than one
 * module in your design. In that case, you should instantiate
 * your top-level module here.
 */
int sc_main( int, char*[] ) {
  my_module top( "top" );
  return 0;
}
```

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 my_project_folder
> fossy my_top_level_design.cc
[...]
```

Listing 1.2: Typographical conventions for “terminal sessions”

2 Using Fossy

2.1 General Information

For accessing and testing the synthesis tool OFFIS operates an online synthesis demo at http://system-synthesis.org/fossy/home. However, the usage of this service is limited to input files of 8000 characters. Moreover, it is limited at the amount of provided synthesis features.

2.2 Structure of Fossy Installation

The installation of *Fossy* is separated into three distinct sub-directories, each containing either binary executables or header files:

Frontend contains a full featured C++ front end, including *cc2cil*, to convert C++ files into a front end specific format, and *cil2xml* to generate XML files for the backend transformation tool.

Fossy contains the main program *fossy* which performs all elaboration and synthesis related tasks.

Library contains the syntheis header files of SystemC.

Although each of the provided tools can be used on their own, the only tool that has to be executed by the user is *fossy*, which is located in *installation_directory/Fossy/bin*. Depending on the type of the given input file (C++, CIL or XML) it automatically calls all other necessary tools. The default behaviour of *fossy* is to take a given SystemC file, execute the front-end, call the CIL-to-XML conversion, load the result and run the synthesis pipeline. As a result, a VHDL-file is generated containing the SystemC design as a synthesisable RT-level design.

2.3 Running Fossy

Before executing *fossy* the user has to set the environment variable `FOSSY_HOME` to the top-level directory of the *Fossy* installation, e.g. for a bash environment:

```
> export FOSSY_HOME=/opt/sw/Fossy
```

The synopsis of *fossy* is:

```
> fossy [OPTION]... <file>
```

Available options are:

```
-h          --help                   Show this help
-V          --version                Show version number and exit
-v          --verbose                Be verbose
-t STRING   --topModule=STRING       Top-level module (if not given, the
                                     elaborator will search for a suitable
                                     one)
-T STRING   --topInst=STRING         Top-level instance name (if not given,
                                     the elaborator generates one)
                                     (not used to search)
-D STRING   --define=STRING          Additional define symbols for frontend
-I STRING   --include=STRING         Additional include paths for frontend
-s          --systemc                Generate SystemC Code
            --noVariableSizeCheck    Disable variable size check
            --noMemStatistics        Disable intermediate memory statistics
-m          --multipleOutputFiles    Write results to multiple output files
                                     (1 file per module)
-a STRING   --aciFile=STRING         ACI file
            --odir=STRING            Set output directory
            --prefix=STRING          Set name prefix for generated files
-k          --keepImplicitFsm        Do not transform implicit to explicit
                                     FSM (Finite State Machine)
-S          --simModel               Generate a pure simulation model
                                     (not for synthesis)
-e          --enumElimination        Eliminate enum types used in top level
                                     ports or signals
-i          --procedureInlining      Inline procedures containing waits into
                                     processes
```

2.4 Example

Listing 2.1 shows a very simple register collection in SystemC. It's a fully parallel register of a parametrised size (and datatype), with one-hot encoded write-enable signals and a synchronous reset. It is implemented with an `SC_METHOD` process for best simulation performance. This is the simplest implementation, since no internal FSM is required. Note that the example uses the *SC_SYNTHESIS* macro to exclude simulation related code from being processed by *Fossy*. If this code has been saved to a file called *example.cpp*, it can be processed by the *Fossy* by entering:

```
> ./fossy example.cpp
```

After *Fossy* has finished the working directory will contain a file called *synthesised_example.vhdl* including the RT-level design.

```
#include <systemc.h>

template< typename DataType, unsigned Size = 8 >
SC_MODULE( sync_register ) {
```

```
  typedef DataType  value_type;

  sc_in<bool> clock;
  sc_in<bool> reset;
  /* ----- input ports ----- */
  sc_in< sc_uint<Size> >      in_wen;
  sc_in< value_type >         in_data;

  /* ----- output ports ----- */
  sc_out< value_type >        data[Size];

  /* ----- constructor ----- */
  SC_CTOR( sync_register ) {
    /* ----- process definitions ----- */
    SC_METHOD( processing );
    sensitive << clock.pos();
  }

private:
  /* ----- process(es) ----- */
  void processing();

}; // sync_register

/* ----- process body ----- */
template< typename DataType, unsigned Size >
void
sync_register<DataType,Size>::processing() {
  sc_uint<10> wen  = in_wen.read();

  for(unsigned i=0; i<Size; ++i) {
    if( reset.read() ) {
      data[i].write( 0 );
    } else if( wen[i].to_bool() ) {
      data[i].write( in_data );
    }
  }
} // sync_register::processing()

/* ----- sc_main() -----
 *
 * We need an explicit main, since we have to
 * instantiate our templated module here.
 */
int sc_main( int, char*[] ) {
  sync_register< sc_uint<32>, 4 > top( "top" );
#ifndef SC_SYNTHESIS
  sc_clock clock("clock", sc_time(10, SC_NS));
  sc_signal< bool > reset;

  my_testbench< sc_uint<32>, 4 > tb( "tb" );
  tb.clk(clock);
  tb.reset(rest)

  top.clock(clock);
  top.reset(reset);
```

```
  top.in_wen(tb.out_wen);
  top.in_data(tb.out_data);
  for(unsigned i=0; i<4; ++i)
    top.data[i](tb.data[i]);

  sc_start(sc_time(2, SC_MS));
#endif
  return 0;
}
```

Listing 2.1: Synchronous Parallel Register Design Example in SystemC

3 Synthesis

3.1 Introduction

Both the internal *Fossy* data structure and the intermediate format of the front end are based on the ISO/IEC C++ standard [cpp98]. All *Fossy* internal code transformations are performed on a C++ standard compliant AST (Abstract Syntax Tree). This approach allows writing out meaningful code after each transformation step enabling traceability and easy debugging.

3.2 Synthesis Subset

This section gives a short overview of the synthesisable SystemC language constructs accepted by the current implementation of the *Fossy* synthesis tool, simply referred to as "the synthesiser".

3.2.1 Compatibility to the SystemC Synthesisable Subset

Fossy is quite compatible to the SystemC Synthesisable Subset. Table 3.1 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.
	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 available.
	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	√	
	Member vars	√	
	Inheritance	√	

	Abstract classes	√	
	Constructors	√	No default arguments allowed.
Overloading		√	
Templates		√	
Pre-processing directives		√	

Table 3.1: *Fossy* SystemC Synthesisable Subset support

3.2.2 Coding Guidelines

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 3.1 and Listing 3.2. 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.

```
#include <systemc.h>
#include <osss.h>

SC_MODULE(SomeModule)
{
  sc_in<bool>    somePort;

  void someProcess();

  SC_CTOR(SomeModule)
  {
    SC_METHOD(someProcess);
    sensitive << somePort;
  }
};
```

Listing 3.1: General structure of the *Fossy* input, `SomeModule.h`

```
#include "SomeModule.h"

void
SomeModule::someProcess()
{
  /* do something useful here */
}
```

Listing 3.2: Source file `SomeModule.cc` corresponding to Listing 3.1

3.2.3 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 3.3.

```
#include <systemc.h>

SC_MODULE(myModule1)
{
  // ...
};

struct myModule2 : public sc_module
{
  // ...

  // Required if this module contains processes
  SC_HAS_PROCESS(myModule2);
};
```

Listing 3.3: A synthesisable module definition

Limitation: Inheritance is not supported for user modules.

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 3.4 and Listing 3.5

```
// Begin of myModule1.h
#include <systemc.h>

SC_MODULE(myModule1)
{
  sc_in<bool>   myPort;

  // Alternative 1:
  // Use SC_CTOR and place the body in the
  // header file
  SC_CTOR(myModule1)
  : myPort("myPort")
  , // ... more initialisers
  {
    /* constructor body */
  }

  // Alternative 2
  // Manually specify the constructor and
  // place the body in the header file
```

```
  myModule(sc_module_name name)
  : myPort("myPort")
  , // ... more initialisers
  {
    /* constructor body */
  }

  // Alternative 3
  // Manually specify the constructor and
  // place the body in the source file
  myModule(sc_module_name);
  // the body is located in the .cc file

  // Alternative 4:
  // Use SC_CTOR and place the body in the
  // source file
  SC_CTOR(myModule1);
  // the body is located in the .cc file
  // (and looks exactly like the one for
  // Alternative 3).

};
// End of myModule1.h
```

Listing 3.4: Synthesisable module constructors, header file

```
// Begin of myModule1.cc:
#include "myModule1.h"

  myModule1::myModule1(sc_module_name)
  : myPort("myPort")
  , // ... more initialisers
  {
    /* constructor body */
  }
// End of myModule1.cc
```

Listing 3.5: 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 complex control structures (if-then-else etc.). Simple (unrollable) for-loops are allowed.

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 3.6.

```
#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 3.6: Synthesisable ports

The type `T` may be any valid data type (see Section 3.2.5).

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 3.2.5).

The instantiation and naming requirements are exactly the same as in the case of ports (Section 3.2.3).
Limitation: 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 3.7.

```
#include <systemc.h>

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

SC_MODULE(Middle)
{
  sc_in<bool>          q1;
  Bottom b;

  sc_signal<int>  s;

  SC_CTOR(Middle)
  : b("b")
  {
    b.p1(q1);    // OK port-to-port binding
    b.p2(s);     // OK port-to-signal binding
                 // NOT OK: unbound b.p3
  }
};

SC_MODULE(Top)
{
  sc_in<sc_uint<8> >   r3;

  Middle m;

  SC_CTOR(Top)
  : m("m")
  {
    m.b.p3(r3); // NOT OK: bypasses the module hierarchy
  }
};
```

Listing 3.7: Synthesisable bindings

Forbidden: Bindings must not bypass modules within the hierarchy.

Limitation: Positional binding is not supported.

3.2.4 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.

Examples are shown in Listing 3.8

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

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

```
#include <systemc.h>

SC_MODULE(Top)
{
  sc_in<bool> clk,
              reset;

  sc_in<int>  px;

  void myMethod();

  // This CTHREAD is intended to show valid and invalid loop
  // constructs (don't mind that the loops are actually unreachable).
  void myCthread()
  {
    int x=4;
    wait();
```

```

    // Valid loop: always runs into a wait each iteration
    while(true)
    {
      if (x--)
        wait(1)
      else
        wait(2);
    }

    // Invalid loop: may do iterations without
    // running into the wait;
    while(true)
    {
      if (x--) wait();
    }

    // Valid loop: always runs into a wait each iteration
    // Note that the wait() be skipped entirely if the condition
    // is false before starting the loop.
    while(x--) wait();

    // Valid loop: always runs into a wait each iteration
    while(true)
    {
      if (x--) wait();

      "Some code";
      wait();
      "More code";
    }

    if(x==0) x=1;

    // OK: some x which is not zero
    wait(x);

    // Valid loop: always runs into a wait each iteration...
    while(true)
    {
      if (x--) wait();

      // ... since this loop always causes a wait();
      for (int i=0;i<3;++i)
      {
        if (i==x-1) wait(); else wait();
      }
    }

    // Valid loop: always runs into a wait each iteration
    do
    {
      wait();
    } while (x--);

  }
```

```
  SC_CTOR(Top)
  {
    SC_METHOD(myMethod);
    sensitive << clk.pos()     // Note: .pos()/.neg() only works
              << reset;        //       for Boolean ports

    sensitive << px;           // OK: there may be multiple
                               //     sensitives

    SC_CTHREAD(myCthread(), clk.pos());
    reset_signal_is(reset, true);
  }
};
```

Listing 3.8: 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[1] 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

[1] 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()`.

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[2] `wait()`s. Note that it is not necessary to test the reset signal in the body of the process. A typical structure is shown in Listing 3.9. 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 save 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`.

```
#include <systemc.h>

SC_MODULE(Top)
{
  sc_in<bool> clk,
              reset;

  sc_in<int>  px;

  void myMethod();

  void myCthread()
  {              //
    int x=4;     // Reset Part
    wait();      //
                    | State1
    while(true)     |     +--+
    {               |     |  |
      x += px;      V     V  | State2
      wait();                |
                          |  |
    }                     +--+
  }

  void myCthread2()
  {              //
    int x=4;     // Reset Part
                 //
    while(true)  //  +--+
    {            //  V  |
      wait();    //     | State1
      x += px;       |  |
    }                +--+
  }

  SC_CTOR(Top)
  {
    SC_CTHREAD(myCthread(), clk.pos());
    reset_signal_is(reset, true);
    SC_CTHREAD(myCthread2(), clk.pos());
    reset_signal_is(reset, true);
  }
```

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

```
};
```

Listing 3.9: Reset part of an `SC_CTHREAD`

3.2.5 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>`,
- `sc_logic` (converted to sc_bv<1>),
- Enumeration types,

Currently not supported are:

- `sc_bit`
- `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 3.2.7.

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.

3.2.6 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 (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";

3.2.7 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 lists
- 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 3.2.5) and member functions must follow the same restrictions as functions do (see Section 3.2.6).

Forbidden: Classes must not have an own thread of control, i.e. any `SC_METHOD`s, `SC_THREAD`s or `SC_CTHREAD`s. These are only allowed in modules.

3.2.8 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.

3.2.9 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.

3.2.10 Polymorphic Objects

Limitation: Polymorphic objects are not yet supported.

3.2.11 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.

3.2.12 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`
- `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)

4 Summary and Support

4.1 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.

4.2 Conclusion

More advanced features and details concerning synthesis can be found in [GBG+08].

References and Further Reading

[cpp98] *ISO/IEC 14882 C++ Standard*, first edition, September 1998. 13

[GBG+08] Kim Grüttner, Claus Brunzema, Cornelia Grabbe, Philipp A. Hartmann, Andreas Herrholz, Henning Kleen, Frank Oppenheimer, Andreas Schallenberg, and Schubert Thorsten. *OSSS - A Library for Synthesisable System Level Models in SystemC^{TM}, The OSSS 2.2.0 Manual*, 2008. 29