R&D Division Transportation
Hardware/Software Design Methodology Group

fossy

Predictable SystemC$^\text{TM}$/C++ Synthesis

The Fossy Manual

Oldenburg 2009
# Contents

1 **Introduction** ................................................................. 5
  1.1 Overview ................................................................. 5
  1.2 Typographical conventions ........................................... 5

2 **Using Fossy** ................................................................. 9
  2.1 General Information ...................................................... 9
  2.2 Structure of *Fossy* Installation .................................... 9
  2.3 Running *Fossy* .......................................................... 9
  2.4 Example ................................................................. 10

3 **Synthesis** ....................................................................... 13
  3.1 Introduction ............................................................... 13
  3.2 Synthesis Subset .......................................................... 13
    3.2.1 Compatibility to the SystemC Synthesisable Subset ....... 13
    3.2.2 Coding Guidelines .................................................. 15
    3.2.3 Design Hierarchy .................................................... 16
    3.2.4 Processes ............................................................. 19
    3.2.5 Datatypes ............................................................. 24
    3.2.6 Statements and Expressions ..................................... 25
    3.2.7 Classes and Inheritance .......................................... 26
    3.2.8 Templates ............................................................ 27
    3.2.9 Namespaces ........................................................... 27
    3.2.10 Polymorphic Objects .............................................. 27
    3.2.11 Shared Objects ..................................................... 27
    3.2.12 Non-Synthesisable ................................................ 27

4 **Summary and Support** ................................................... 29
  4.1 Support ................................................................. 29
  4.2 Conclusion ............................................................... 29

**References and Further Reading** ........................................ 29
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 Functional Oldenburg System SYnthesiser.
- **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 following 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 starts with a general introduction followed by the definition of the supported synthesis subset.
- Chapter 4 provides contact information in case of technical questions and recommends 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.
<table>
<thead>
<tr>
<th>Convention</th>
<th>Item</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times New Roman</td>
<td>Normal Text</td>
<td>This is an example sentence.</td>
</tr>
<tr>
<td>Times New Roman</td>
<td>Emphasised Text</td>
<td>This <em>word</em> is emphasised.</td>
</tr>
<tr>
<td>Monospace font</td>
<td>Class, function, method or macro names</td>
<td><code>execute_operation()</code></td>
</tr>
<tr>
<td>Monospace italics</td>
<td>Variables meant to be replaced when the language construct is used.</td>
<td><code>my_class&lt;parameter&gt;</code></td>
</tr>
<tr>
<td></td>
<td>Sometimes variables or parameters are omitted.</td>
<td><code>my_class&lt;...&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>do_something(...)</code></td>
</tr>
<tr>
<td>Monospace font</td>
<td>Shell commands</td>
<td><code>make</code></td>
</tr>
</tbody>
</table>

Table 1.1: Typographical conventions for continuous text
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.
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 synthesis 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:

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

The synopsis of *fossy* is:

```bash
> 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:

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

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

```c++
#include <systemc.h>

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

sc_in<sc_uint<Size>> in_wen;
sc_in<value_type> in_data;

sc_in<bool> clock;
sc_in<bool> reset;

sc_out<value_type> data[Size];

typedef struct{
    void processing();
} sync_register;

/∗ − − − − − process (es) − − − − − ∗/
void sync_register<DataType, Size>::processing() {
    for(unsigned i=0; i<Size; ++i) {
        if( reset.read() ) {
            data[i].write( 0 );
        } else if( in_wen.read() ) {
            data[i].write( in_data);
        }
    }
}

/∗ − − − − − sc_main ( ) − − − − − ∗/
int sc_main( int , char [] ) {
    my_testbench<sc_uint<32>, 4 > tb( "tb" );
    sc_clock clock("clock", sc_time(10, SC_NS));
    sc_signal<bool> reset;
    top.clock(clock);
    top.reset(reset);
    return 0;
}
Using Fossy

Listing 2.1: Synchronous Parallel Register Design Example in SystemC

```c

    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;
```


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.

<table>
<thead>
<tr>
<th>C++/SystemC™ feature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation units</td>
<td>√</td>
</tr>
<tr>
<td>Only one translation unit allowed.</td>
<td></td>
</tr>
<tr>
<td>Modules</td>
<td></td>
</tr>
<tr>
<td>Definition: SC_MODULE, sc_module inheritance</td>
<td>√</td>
</tr>
<tr>
<td>Members: signal, sub-module, ctors, HAS_PROCESS</td>
<td>√</td>
</tr>
<tr>
<td>Ports/Signals: sc_signal, sc_in, sc_out, sc_inout, sc_in_clk</td>
<td>√</td>
</tr>
<tr>
<td>Ports/Signals: sc_out_clk, sc_inout_clk</td>
<td>–</td>
</tr>
<tr>
<td>Not supported, but deprecated anyway.</td>
<td></td>
</tr>
<tr>
<td>Ports/Signals: *_resolved, *_rv</td>
<td>–</td>
</tr>
<tr>
<td>Not supported.</td>
<td></td>
</tr>
<tr>
<td>Ctor: w/ and w/o SCCTOR macro</td>
<td>√</td>
</tr>
<tr>
<td>Deriving</td>
<td>√</td>
</tr>
<tr>
<td>Datatypes</td>
<td></td>
</tr>
<tr>
<td>Integral types</td>
<td>√</td>
</tr>
<tr>
<td>Integral promotion, arith. conversion</td>
<td>√</td>
</tr>
<tr>
<td>Operators</td>
<td>√</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Compounds: arrays, enums, class/struct/unions, functions</td>
<td>√ No pointers and no references supported.</td>
</tr>
<tr>
<td>SysC: \texttt{sc_int}, \texttt{sc_uint}, \texttt{sc_bigint}, \texttt{sc_biguint}</td>
<td>√</td>
</tr>
<tr>
<td>SysC: fixed-point types</td>
<td>– Not supported.</td>
</tr>
<tr>
<td>SysC: \texttt{sc_bv}</td>
<td>√</td>
</tr>
<tr>
<td>SysC: \texttt{sc_logic}</td>
<td>√ Converted to \texttt{sc_bv&lt;1&gt;} internally.</td>
</tr>
<tr>
<td>SysC: \texttt{sc_lv}</td>
<td>–</td>
</tr>
<tr>
<td>SysC: arithmetic operators</td>
<td>√</td>
</tr>
<tr>
<td>SysC: bitwise operators</td>
<td>√</td>
</tr>
<tr>
<td>SysC: relational operators</td>
<td>√</td>
</tr>
<tr>
<td>SysC: shift operators</td>
<td>√</td>
</tr>
<tr>
<td>SysC: assignment operators</td>
<td>√ No chained assignments, LHS must be side effect-free.</td>
</tr>
<tr>
<td>SysC: bit select operators</td>
<td>√</td>
</tr>
<tr>
<td>SysC: part select operators</td>
<td>√ Reverse ranges not supported.</td>
</tr>
<tr>
<td>SysC: concatenation operators</td>
<td>√ Assignment to concats not supported.</td>
</tr>
<tr>
<td>SysC: conversion to C integral (\texttt{to_int()}, \texttt{to_bool()} etc.)</td>
<td>√</td>
</tr>
<tr>
<td>SysC: additional methods: \texttt{iszero()}, \texttt{sign()}, \texttt{bit()}, \texttt{reverse()}, etc.</td>
<td>– accepted, but not fully implemented</td>
</tr>
<tr>
<td>Declarations</td>
<td></td>
</tr>
<tr>
<td>typedef</td>
<td>√</td>
</tr>
<tr>
<td>enums</td>
<td>√</td>
</tr>
<tr>
<td>aggregates</td>
<td>– Not supported. Reduced support for ROM initialization available.</td>
</tr>
<tr>
<td>arrays</td>
<td>√</td>
</tr>
<tr>
<td>references</td>
<td>– Copy-in copy-out support for functions/methods.</td>
</tr>
<tr>
<td>pointers</td>
<td>– Pointers allowed for sub-module instantiation.</td>
</tr>
<tr>
<td>Expressions</td>
<td>√ new/delete/pointers not supported, no chained assignments. \texttt{new} allowed for sub-module instantiation.</td>
</tr>
<tr>
<td>Functions</td>
<td>√</td>
</tr>
<tr>
<td>Statements</td>
<td>√</td>
</tr>
<tr>
<td>Processes</td>
<td>√ No \texttt{SC_THREAD} supported.</td>
</tr>
<tr>
<td>Sub-module instantiation</td>
<td>√ No support for positional port binding.</td>
</tr>
<tr>
<td>Namespaces</td>
<td>√</td>
</tr>
<tr>
<td>Classes</td>
<td></td>
</tr>
<tr>
<td>Member functions</td>
<td>√</td>
</tr>
<tr>
<td>Member vars</td>
<td>√</td>
</tr>
<tr>
<td>Inheritance</td>
<td>√</td>
</tr>
</tbody>
</table>
3.2 Synthesis Subset

<table>
<thead>
<tr>
<th>Feature</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract classes</td>
<td>✓</td>
</tr>
<tr>
<td>Constructors</td>
<td>✓ No default arguments allowed.</td>
</tr>
<tr>
<td>Overloading</td>
<td>✓</td>
</tr>
<tr>
<td>Templates</td>
<td>✓</td>
</tr>
<tr>
<td>Pre-processing directives</td>
<td>✓</td>
</tr>
</tbody>
</table>

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.

```c++
#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

```c++
#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.

```cpp
#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

```cpp
// 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 SCCTOR and place the body in the
// source file
SCCTOR(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

#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>`
• `oss_s_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 initialiser list or in the constructor body as shown in Listing 3.6.

```cpp
#include <systemc.h>

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

  SCCTOR(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;

    SCCTOR(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;

    SCCTOR(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_CTHREADs:
• **SC_CTHREADs** 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_CTHREADs** 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_CTHREADs**):

• 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.

```cpp
#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();
  }
```
3.2 Synthesis Subset

// 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--);
Effect of \texttt{wait()} usage on the number of states

In general the number of states of the resulting state machine equals the number of \texttt{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 \texttt{wait()}s, \texttt{Fossy} tries to unroll that loop. Consequently the number of resulting states is the number of \texttt{wait()} statements times the number of iterations. Please keep in mind that loops with many iterations which have to be unrolled due to non-optimal \texttt{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 \texttt{wait(n)}s. There are two different synthesis strategies. The first one is to simply unroll the \texttt{wait(n)} by \texttt{n wait(1)}s which results in \texttt{n} states. The second one is to replace the \texttt{wait(n)} by a loop with an unconditional \texttt{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 \texttt{n} is a compile-time known constant and also on its value.

Reset

The reset signal of an \texttt{SC_CTHREAD} is determined by the correpsonding \texttt{reset_signal_is(\ldots)} statement in the constructor body. Please note that the reset of an \texttt{SC_CTHREAD} is always synchronous to the \texttt{SC_CTHREAD}'s clock. Consequently, \texttt{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 \texttt{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 \texttt{SC_CTHREAD} would mean to start at some arbitrary \texttt{wait()}. In other words, the SystemC simulation with \texttt{SC_CTHREAD}s suggests that the state machine starts in its initial state even

\footnote{In this case the \texttt{wait()}s will be conditional ones, because otherwise the loop would not have a path through its body without encountering a \texttt{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`.

\(^2\)There may be more than one possible first `wait()` in the case of conditional `wait()`s in the reset part.
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 \texttt{operator==(...) and operator<<(...)} for stream insertion (and a \texttt{sc_trace(...) function})

• Signal-ports (sc_in<T>, sc_out<T>, sc_inout<T>)

• typedef

\textit{Limitation:} The resolved signal \texttt{sc_signal_rv<N>} and the corresponding ports \texttt{sc_in_rv<N>, sc_out_rv<N>} and \texttt{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 \texttt{if () else, switch, for} and \texttt{while} loops are synthesisable. It is allowed to have \texttt{switch} statements with fall-through cases, i.e. cases without a \texttt{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 \texttt{const} or non-\texttt{const}.

\textit{Forbidden:} Functions and operators must not return references.

\textit{Limitation:} Chaining multiple \texttt{operator =} in assignments (such as \texttt{a=b=c=d=0;}} is not supported, yet.

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

\textit{Limitation:} A variable declaration in a switch-block before the first case label is not supported.

\textit{Limitation:} Non-toplevel case labels are not supported.

\textit{Limitation:} Variable declarations in switch clauses without a surrounding block are not supported.

\textit{Limitation:} A non-void routine must not contain a return statement without an expression.

\textit{Limitation:} Recursion is not supported.

\textit{Limitation:} sizeof(...) is not supported.

\textit{Limitation:} Only SystemC bitvector literals with prefix "0b0" are supported, e.g. \texttt{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_METHODs, SC_THREADs or SC_CTHREADs. 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 synthesizable:

- 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
