

Confidential




**IST-5- 033511**

**ANDRES**

**ANalysis and Design of run-time REconfigurable,  
heterogeneous Systems**

Project Duration 2006-06-01 – 2009-05-31

Type STREP

	WP no.	Deliverable no.	Lead participant
	<b>WP1</b>	<b>D1.5b</b>	<b>TUV</b>
<b>Modelling extensions for polymorphic signals, final library elements</b>			
Prepared by	<b>Markus Damm, Joseph Wenninger - TUV</b>		
Issued by	<b>TUV</b>		
Document Number/Rev.	<b>ANDRES/TUV/P/D1.5b/1.0</b>		
Classification	<b>ANDRES Confidential</b>		
Submission Date	<b>2008-11-30</b>		
Due Date	<b>2008-11-30</b>		
<b>Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)</b>			

© Copyright 2006 .Diseno de Sistemas en Silicio S.A., Kungliga Tekniska Högskolan, Kuratorium OFFIS, Vienna University of Technology, Universidad de Cantabria, Thales Communications S.A

This document and the information contained herein may not copied, used or disclosed in whole or in part outside of the consortium except with prior written permission of the partners listed above.

## Contents

1	Introduction	3
2	Usage	4
3	Internal representation	6
4	Implementation	8
5	MoC conversion	11
5.1	Conversion from SDF	12
5.1.1	SDF to FIFO	12
5.1.2	SDF to SC	13
5.1.3	SDF to ELEC_VOLTAGE/ELEC_CURRENT	13
5.2	Conversion from FIFO	14
5.2.1	FIFO to SDF	14
5.3	Conversion from SC	15
5.3.1	SC to SDF	15
5.3.2	SC to FIFO	15
5.3.3	SC to ELEC_VOLTAGE/ELEC_CURRENT	16
5.4	Conversion from ELEC_VOLTAGE/ELEC_CURRENT	17
5.4.1	ELEC_VOLTAGE/ELEC_CURRENT to SDF	17
5.4.2	ELEC_VOLTAGE/ELEC_CURRENT to SC	17
5.4.3	ELEC_VOLTAGE/ELEC_CURRENT to FIFO	18
6	Data type conversion	19
7	Current distribution	23
8	Conclusion and future work	23
9	References	24

## 1 Introduction

The purpose of the converter channels (as they have been renamed to from polymorphic signals [1], see also [2]) within the ANDRES-project [3] is the connection of modules/processes lying within the domains of different Models of Computation (MoCs). The motivation is twofold:

### **Mixed level Simulation [4]**

When following a design methodology based on design refinement, a refinement step for a certain system part often involves changing the MoC for this part. For example, a filter might be described behaviourally within the timed data flow (TDF) MoC, and after refinement, it has an analogue representation as an electrical network (CT-NET MoC), or one as a digital filter using the discrete event (DE) MoC. However, the designer might not want to change the MoCs for the other system parts, but simulate the system with certain parts being in different refinement levels. This task requires the insertion of MoC converters with respect to the different MoCs used. The designer might also want to change the refinement levels of different system parts back and forth, for example to save simulation time by leaving most system parts on an abstract system level which can be simulated fast, while only describing certain system parts of interest on a more detailed, nearer to implementation level, which costs more simulation performance. This requires repetitive manual insertion and removal of MoC converters, which is a laborious and error prone task.

### **Design space exploration**

When a designer investigates the different possible implementations for a heterogeneous system, there are usually several possibilities to implement certain system parts. Many system parts can be implemented using software, digital hardware, or analogue hardware. Also, the communication means might be subject to change during the exploration process, e.g. by varying bit widths.

Switching between two design alternatives often involves changing the MoC of certain system parts, which again requires the repetitive use of different MoC converters. Here, we might also face the problem of data type conversion, since the data types used might also change, for example when switching from an analogue filter with a continuous output (e.g. `double`) to a digital filter with a digital output, whereas the bit width used in the latter case might also be varied. In certain cases, a converter channel might even take the role of an analogue/digital converter (or vice versa), and by using different system partitions, the designer might shift this role from one converter channel to another.

Converter channels ease the process of inserting and removing MoC and/or data type converters. They are general purpose channels, which can be used to connect system parts using different MoCs and data types. The conversion means necessary are hidden within the channel, and can be steered by the designer with a few parameters/options. Moreover, the conversion means are mostly provided automatically, e.g. when the MoC of a system part that reads from a converter channel changes, the designer has mostly not to make any changes to the converter channel. If there are problems with the conversion process, e.g. an overflow when doing data type conversion, or certain corner cases which can occur for certain MoC conversion directions, the designer is provided with appropriate warnings.

In this report, we describe the final implementation of the converter channel class for the ANDRES-project regarding usage and implementation.

## 2 Usage

Technically, a converter channel is a C++ class named `converterchannel`, which derives from the SystemC class `sc_core::sc_channel`. It is placed within a separate namespace called `ahes`. A converter channel is instantiated as follows:

```
ahes::converterchannel<M, DT_W, DT_R1, DT_R2> conv;
```

where

- `M` denotes the MoC of the writing side (mandatory)
- `DT_W` denotes the data type of the writing side (mandatory)
- `DT_R1` and `DT_R2` denote the possible data types of the reading sides of this converter channel (optional)

If only `DT_W` is set, the converter channel won't offer any data type conversion capabilities, and all (reading or writing) ports bound to it have to be of data type `DT_W`. In general, the writing port bound to the converter channel has to be of data type `DT_W`, and any reading port bound to it has to be of one of the data types `DT_W`, `DT_R1` or `DT_R2`. Note that `DT_W`, `DT_R1` and `DT_R2` have to be pairwise distinct. In the initial version of the converter channels (see D1.5a), up to five different reading side data types could be used. However, this presented us with several implementation difficulties regarding code maintenance, and we decided to reduce this number to two, after consulting with our industrial partners, who agreed that this number should be sufficient.

The MoC `M` is specified with an enum type `ahes::MoC`, which has the following members:

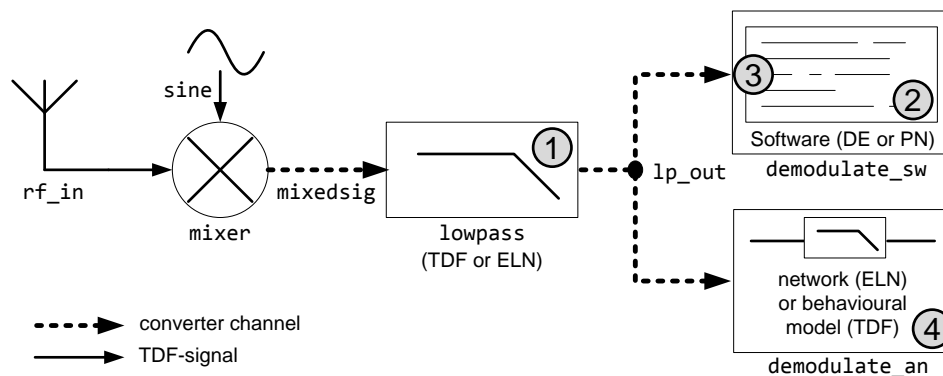
- `ahes::SC` denotes the *Discrete Event* (DE) MoC of SystemC
- `ahes::SDF` denotes the *Timed Synchronous Data Flow* (TDF) MoC of SystemC-AMS
- `ahes::FIFO` denotes the *Kahn Process Network* (KPN) MoC
- `ahes::ELEC_VOLTAGE` and `ahes::ELEC_CURRENT` denote the *continuous time electrical network* (CT-NET) MoC of SystemC-AMS, depending on the electrical quantity of interest

From the coding point of view, a converter channel can be used almost like a signal in SystemC. Binding a converter channel to module ports works like this:

- `mod_write.out(conv)` for the writing port `out` of a module `mod_write`
- `mod_read.in(conv)` for the reading port `in` of a module `mod_read`

Note that binding to the writing side now works the same way as binding to the reading side, in contrast to the initial version of the converter channels, where a special method indicating the writing side MoC had to be called for connecting to the writing side. By using a template parameter to indicate the writing side MoC of a converter channel, we could eliminate this shortcoming.

Before explaining the different capabilities of the converter channels more details, we give an example on how converter channels can be used with respect to mixed level simulation and design refinement. The example deals with the evaluation of a software defined radio (SDR). An overview of the system is shown in Figure 2.1. The RF input signal is mixed with a sine wave which has the same frequency as the carrier signal and the result is processed by a lowpass filter. After that, the demodulation is done by software and an analogue demodulator, to compare the results. The input of the software demodulator is an integer with fixed bit width. Listing 2.1 shows the corresponding top-level SystemC code using two converter channels. Note that the code example assumes the `ahes` namespace to be included. The converter channel method `setRangeScaling` in line 6 is used to steer quantization, and will be explained in more detail later.



**Figure 2.1.** A refinement example: SW-defined radio

```

bitwidth=8;
sca_sdf_signal<double> rf_in;           // incoming RF-signal
sca_sdf_signal <double> sine;          // sine wave

converterchannel<TDF, double> mixedsig; // RF-signal multiplied with sine-wave
converterchannel<TDF,double, sc_int<bitwidth> > lp_out; // output of lp-filter
lp_out.setRangeScaling<sc_int<bitwidth> > (-1. , 1. ); // assuming the value
// range within [-1,1]

mixer mix("mix"); // mixes the two input signals
  mix.in1(rf_in);
  mix.in2(sine);
  mix.out(mixedsig);

lowpass_behavioural lp("lp");           // lowpass filter, either T-SDF-module
  lp.in(mixedsig);                      // or electrical network
  lp.out(lp_out);

demodulate_sw dem_sw("dem_sw", bitwidth); // software demodulator with
  dem_sw.in(lp_out);                    // sc_int<bitwidth> input

demodulate_an dem_an("dem_an");        // analogue demodulator
  dem_an.in(lp_out);

```

**Listing 2.1:** SW-defined radio with converterchannels

Regarding design space exploration and mixed level simulation, this example gives rise to the following tasks (the numbers refer to those in Figure 8):

1. Realizing the lowpass filter either as a (behavioural) TDF-module or as an electrical network.

2. Realizing the software demodulator either as a kahn process network (KPN) or as a DE module
3. Varying the bit width of the input of the software demodulator
4. Realizing the analogue demodulator either as a (behavioural) TDF-module or as an electrical network.

To make the matter more complicated, any subset of these tasks can be done in parallel. It is obvious, that the effort for manual inserting (and adapting) the appropriate converters would be significant. Let's assume an initial model with the lowpass filter and the analogue demodulator modeled as T-SDF modules and the software demodulator modelled within the KPN MoC, taking `sc_int<bitwidth>` inputs. We then would need a TDF→KPN converter from the output of the lowpass filter to the software modulator, which would also convert double values to `sc_int<bitwidth>` values. Now, executing the tasks above would call for the following respective manual conversion activities:

1. Realizing the lowpass filter as an electrical network: Insert a TDF→CT-NET converter between the mixer and the lowpass filter. Replace the initial TDF→KPN converter by a CT-NET→KPN converter, which also converts double to `sc_int<bitwidth>`. Instantiate appropriate signals to connect them.
2. Realizing the software demodulator as a DE module: Replace the initial TDF→KPN converter by a TDF→DE converter, which also converts double to `sc_int<bitwidth>`. Note that using a SystemC AMS converter port makes no sense here, since the analogue demodulator is still in the TDF domain. Instantiate appropriate signals to connect them.
3. Varying the bit width of the input of the software demodulator: Change the output of the TDF→KPN converter and the type of the signal which connects it to the software demodulator to `sc_int<new_bitwidth>`. The data type conversion algorithm of the converter must also be altered slightly. Of course, using a parametrizable converter regarding the bitwidth would make things easier here. In that case, this design space exploration task could be steered similar to the code example above.
4. Realizing the analogue demodulator as an electrical network: Insert a TDF→CT-NET converter between the lowpass filter and the analogue demodulator. Instantiate an appropriate signal to connect them.

By using converter channels, however, the code for each of the possible variants would be very similar to the code in Example 3: The value of the variable bitwidth would change, as well as the class names of the respective modules (e.g. changing `lowpass_behavioural` to `lowpass_electrical`). In some cases, the writing side MoC of a converter channel has to be changed. This very simple example shows the convenience converter channels offer to the designer.

### 3 Internal representation

In this section, we give some detail on how the code described above is processed, i.e. handled internally in terms of signals and converter modules. In principle, if a converter channel connects two modules `m_write` and `m_read` having a port `out` and `in` respectively, it instantiates a signal which fits to the port `out` (the *input signal*), a signal which fits to the port `in` (the *output signal*), and a converter which reads the input signal and process it in a way that it can be written to the output signal.

Figure 3.1 shows an example: `m_write` is a SystemC discrete event (DE) module having a port `out` of type `sc_out<double>`, and `m_read` is a SystemC-AMS synchronous dataflow (TDF) module having a port `in` of type `sca_sdf_in<int>`. The SystemC-AMS library provides `sca_scsdf_in` and `sca_scsdf_out` ports, which can be used within TDF-modules to connect to DE-signals (i.e. `sc_signals`). Therefore, a converter module from DE to TDF can be realized by a TDF-module having an `sca_scsdf_in` input and a `sca_sdf_out` output. Then it can be connected to `m_write` and `m_read` using the input and output signal, respectively. The only action remaining within the converter module in this case is data type conversion, which is done by calling the appropriate conversion function.

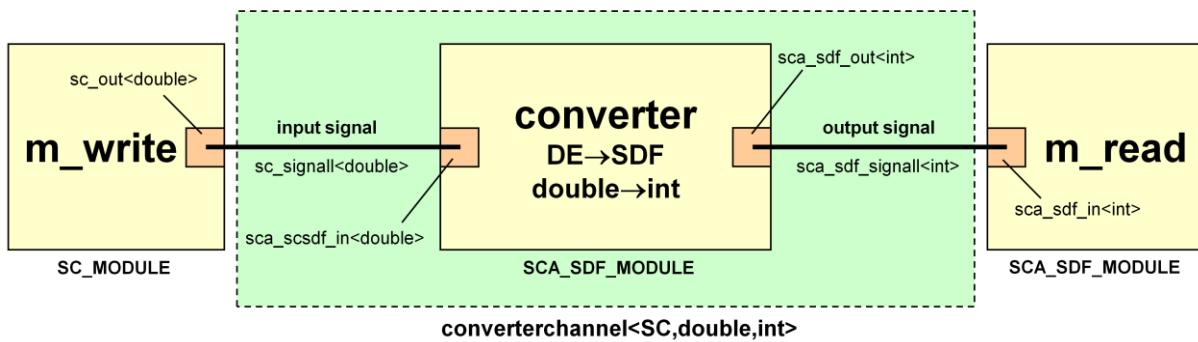


Figure 3.1: example of the internal structure of a polymorphic signal

In general, not every conversion capability is needed. If the data types on both sides agree, only MoC conversion is performed, and if the MoCs agree, only data type conversion is performed. If data types as well as MoCs agree, `m_write` and `m_read` are directly connected with the input signal.

In the general case, a converter channel will be connected to several readers. Here, apart from the input signal, several converters and output signals are created. Each converter is then connected to the input signal and to a suited output signal. Each reading module is connected to the appropriate output signal. If a reading module happens to agree to the writing module regarding MoC and data type, it is directly connected to the input signal. Figure 3.2 shows an example.

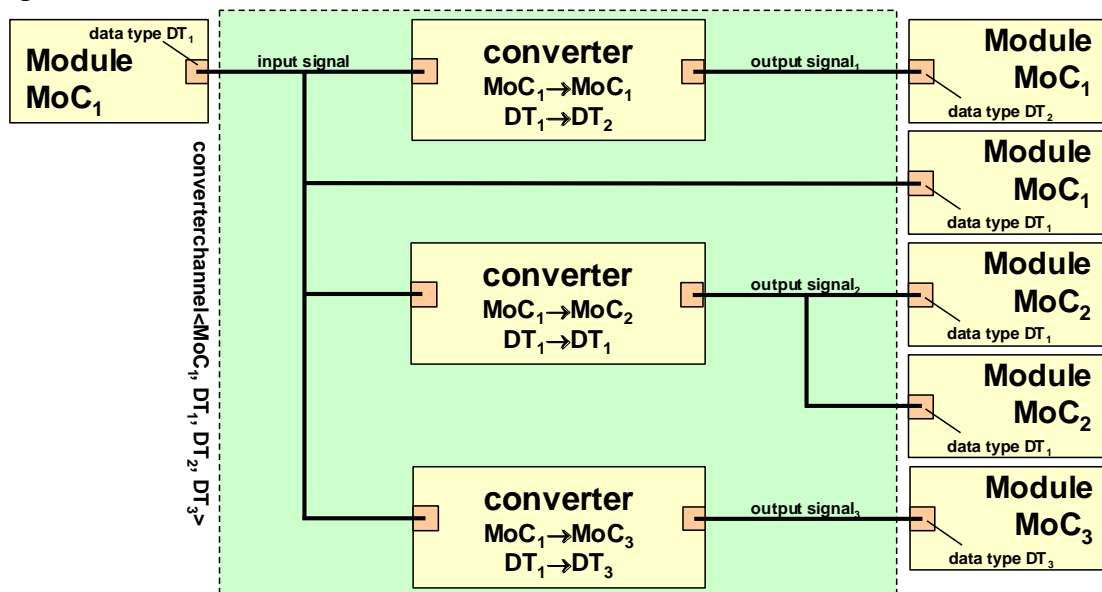


Figure 3.2: internal structure of a converter channel connected to multiple reading modules

In principle, every conversion case is handled the way described above. The only cases which are treated slightly different are those when electrical nodes are involved. This will be explained in more detail later.

## 4 Implementation

The `converterchannel` class is a templated class with two mandatory template parameter (the enum parameter `MOC` and the data type `T_WRITE`, which is the data type of the writing port) and two optional template parameters `T_READ1` and `T_READ2`, the data types of additional reading ports, which are set to dummy-classes by default. The data of the class consist mainly of pointers to different in- and output signals and converters. Listing 4.1 shows the code, which lies within the `ahes` namespace:

```
template <MoC MOC_WRITE, typename T_WRITE, typename T_READ1=impl::dummy<1>,
typename T_READ2=impl::dummy<2> >
class converterchannel : public sc_core::sc_channel{
public:

    ahes_signal<MOC_WRITE,T_WRITE>* insignal;        // the input signal

    sc_signal<T_WRITE>* outsignal_sc_0;            // pointers for sc_signals
    sc_signal<T_READ1>* outsignal_sc_1;            // to be passed to reading ports
    sc_signal<T_READ2>* outsignal_sc_2;            // one for each data type

    sca_sdf_signal<T_WRITE>* outsignal_sdf_0;        // The same as above for sdf
    sca_sdf_signal<T_READ1>* outsignal_sdf_1;
    sca_sdf_signal<T_READ2>* outsignal_sdf_2;

    sc_fifo<T_WRITE>* outsignal_fifo_0;            // The same as above for sc_fifos
    sc_fifo<T_READ1>* outsignal_fifo_1;
    sc_fifo<T_READ2>* outsignal_fifo_2;

    sca_elec_node* internal_elec_node_pos;        // the internal
    sca_elec_node* internal_elec_node_neg;        // electrical nodes

    converter<MOC_WRITE,SC,T_WRITE,T_WRITE>* conv_2sc_0;        // pointer to
    converter<MOC_WRITE,SC,T_WRITE,T_READ1>* conv_2sc_1;        // converters
    converter<MOC_WRITE,SC,T_WRITE,T_READ2>* conv_2sc_2;
    converter<MOC_WRITE,SDF,T_WRITE,T_WRITE>* conv_2sdf_0;
    converter<MOC_WRITE,SDF,T_WRITE,T_READ1>* conv_2sdf_1;
    converter<MOC_WRITE,SDF,T_WRITE,T_READ2>* conv_2sdf_2;
    converter<MOC_WRITE,FIFO,T_WRITE,T_WRITE>* conv_2fifo_0;
    converter<MOC_WRITE,FIFO,T_WRITE,T_READ1>* conv_2fifo_1;
    converter<MOC_WRITE,FIFO,T_WRITE,T_READ2>* conv_2fifo_2;
    converter<MOC_WRITE,ELEC_VOLTAGE,T_WRITE,double>* conv_2voltage;
    converter<MOC_WRITE,ELEC_CURRENT,T_WRITE,double>* conv_2current;
    // the converters to ELEC_X always have reading data type double

    converterchannel_options<T_WRITE, T_WRITE>* options_0; // options to be
    converterchannel_options<T_WRITE, T_READ1>* options_1; // passed to the
    converterchannel_options<T_WRITE, T_READ2>* options_2; // converters
    converterchannel_options<T_WRITE, double>* options_D;

    // Data for special cases and optional capabilities:
    MoC elec_mode; // determines if CT-NET conversion uses voltage or current
    double elec_scaling_factor; // stores gain of CT-NET conversion
```



```

int readingFIFOSize; // the size of the internal fifo on the reading side
sc_signal<bool>* sdf_clk; // clock-signal that runs with the speed of
                        // the SDF input cluster (if any)
sdf_clock* sdf_clk_gen; // module producing this clock-signal
lost_val_alerter<T_WRITE>* alerter; // checks for SC-SDF undersampling

```

**Listing 4.1:** the class data of the converter channel

The `ahes_signal` class is a template class with specializations for each MoC that inherit from the appropriate class. For example, an instance of `ahes_signal<SC, int>` is simply an `sc_signal<int>`, while a `ahes_signal<ELEC_VOLTAGE, double>` is an `elec_node`. That way, only one pointer is needed to hold the input signal. There are various pointers for each possible output signal, but a respective instance is only created when needed; the same holds for the converters. The `converter` class is a template class with template parameters the writing side MoC, the reading side MoC, as well as the writing and the reading side data types. For each MoC pair, there exists an appropriate specialization of the converter class, which then inherits the necessary data type capabilities from an appropriate data type conversion class. For example, the header of the TDF to DE converter looks like in Listing 4.2, where the `datconv` class is the data type conversion class that provides the data type conversion function.

```

template <typename T_WRITE, typename T_READ>
class converter<SDF, SC, T_WRITE, T_READ>:
    sca_sdf_module, public datconv<T_WRITE, T_READ>

```

**Listing 4.2:** example converter header

The struct `converterchannel_options` in Listing 4.1 is used to pass several options to the converters. Some of the options are data type specific, but also the MoC specific options are included. It is only used internally, and will not be set by the user. The rest of the class data refers to special options for certain MoC conversion cases, which will be explained later.

The connection of a `converterchannel` to ports is achieved by overloading the bracket-operator. Listing 4.3 shows the corresponding method for the connection to `sc_in/sc_out` ports.

```

operator sc_signal<T_WRITE>& () {
    if(MOC_WRITE!=SC) { // => must be reading side connection
        if(outsignal_sc_0==0) { // we have no SC-reader connected yet?
            outsignal_sc_0 = new sc_signal<T_WRITE>; // => create the signal
        }
        return *outsignal_sc_0;
    }
    else
    {
        writer_connected = true;
        return *(dynamic_cast<sc_signal<T_WRITE>*>(insignal));
    }
}

```

**Listing 4.3:** ()-operator to connect the converter channel to a reading SC-port

In the `before_end_of_elaboration()` method, which is called for every SystemC-module before the simulation starts, the required converters are instantiated and connected to the appropriate signals. For example:

```

if(outsignal_sc_0!=0 && MOC_WRITE != SC){
    conv_2sc_0 =
        new converter<MOC_WRITE,SC,T_WRITE, T_WRITE>("conv_2sc_0", options_0);
    conv_2sc_0->in(*insignal);
    conv_2sc_0->out(*outsignal_sc_0);
}

```

**Listing 4.4:** setting up the converters

As an example, the code for the converter from TDF to DE is shown in Listing 4.5. In this MoC conversion case, the user has the opportunity to get a clock signal which runs with the same speed the TDF port is sampled. From the user's side, this behaviour is triggered by accessing the method `clock_sdf()` of the converter channel, which then sets the converter channel's options objects accordingly. Apart from that, the conversion work is done by the converter port out, such that in the `sig_proc()` method, the only thing to do (apart from starting the `sdf_clock_gen` module) is reading the input value, converting it, and writing it to the converter port.

```

template <typename T_WRITE, typename T_READ>
class converter<SDF, SC, T_WRITE, T_READ>:
    sca_sdf_module, public datconv<T_WRITE, T_READ>{
public:
    sca_sdf_in<T_WRITE> in;    // TDF in-port
    sca_scsdf_out<T_READ> out; // converter port SC <-> SDF

    sdf_clock* sdf_clk_gen;    // clock with speed of in's sampling period
    bool clock_started, using_sdf_clock;

    converter(sc_module_name n, //constructor
              converterchannel_options<T_WRITE, T_READ>* options
              ) : datconv<T_WRITE, T_READ>(options)
    {
        if(options->sdf_clk_gen!=0){ // clock is used
            sdf_clk_gen=options->sdf_clk_gen;
            using_sdf_clock = true;
            clock_started=false;
        }
        else // clock is not used
        {
            using_sdf_clock = false;
            clock_started=true;
        }
    }

private:
    void sig_proc() {
        out.write(conv(in.read()));
        if(!clock_started && using_sdf_clock)
        {
            clock_started = true;
            sdf_clk_gen->start(in.get_T());
        }
    };
};

```

**Listing 4.5:** The SC to TDF converter

The *data type conversion* capabilities are provided by the templated super class `datconv<T_WRITE, T_READ>`, from which all converters inherit. It provides the

`conv()`-method, which converts from the data type `T_WRITE` to the data type `T_READ`. The `datconv` class is mainly defined through specialized classes, but there's also a general one which illustrates the idea:

```
template <typename T_WRITE, typename T_READ>
class datconv{
public:
    datconv(converterchannel_options<T_WRITE, T_READ>* options){};

    T_READ conv(T_WRITE x){
        return static_cast<T_READ>(x);
    };
};
```

Listing 4.6: The general data type converter

The specialized data type converters deal with those cases, where casting is not sufficient, e.g. when overflow might be an issue, or to handle certain data type conversion options.

## 5 MoC conversion

Table 5.1 shows which MoC conversion directions are supported by the converter channels. The conversion between `ELEC_VOLTAGE` and `ELEC_CURRENT` is not supported, since this conversion is too delicate to be hidden within a channel, while there is also no obvious application for such a converter channel conversion. Conversions like this should be implemented with the according SystemC AMS facilities, like voltage controlled current sources.

There has not been implemented a conversion from `FIFO` other than then `FIFO` to `SDF` conversion. The reason is that the semantics of a conversion from `FIFO` to e.g. `SC` is unclear. It is necessary for this conversion direction to provide an internal `sc_fifo` channel on the writing side, otherwise the process writing to the converter channel could not block, which might lead to an endless loop. Since the `SC` side does not read actively in general, the converter channel would need to empty the internal fifo by itself, e.g. due to a time interval passed to the converter channel; this would effectively yield the same conversion semantics as `FIFO` to `SDF` conversion. This issue needs some discussion before implementing this conversion direction.

from \ to	SC	SDF	FIFO	ELEC_VOLTAGE	ELEC_CURRENT
SC		✓	✓	✓	✓
SDF	✓		✓	✓	✓
FIFO	?	✓		?	?
ELEC_VOLTAGE	✓	✓	✓		✗
ELEC_CURRENT	✓	✓	✓	✗	

Table 5.1: Overview on the MoC conversion capabilities

Although every conversion direction described is working, there is an **important constraint regarding the use of converter channels and electrical networks**. Although electrical networks are treated as a MoC of its own right, they can be handled in SystemC AMS only with the use of `SDF`. That is, every electrical network has to be connected to e.g. an `SDF`-controlled voltage source or e.g. a current to `SDF` converter.

Therefore, if an electrical network is not connected to an SDF cluster anyway, e.g. by the means described above, it is necessary to include at least one to or from SDF conversion with a converter channel. For example, it would be sufficient to connect an electrical network to a `converterchannel<SDF,->`, but connecting it to a `converterchannel<SC,->` and a `converterchannel<ELEC_VOLTAGE,->`, where the latter converter channel does not connect to an SDF cluster, would not be sufficient.

In the rest of this section the semantics of every MoC conversion is described shortly, including possible corner cases and options. For simplicity, we refer to the identifiers in the enum type `MoC` for the different MoCs.

## 5.1 Conversion from SDF

### 5.1.1 SDF to FIFO

In this case, we have a SDF writer and a reader, which expects to read from a `sc_fifo<>` channel. Here, the `converterchannel` instantiates an internal `sca_sdf_signal<T_WRITE>` for the writing side and a `sc_fifo<T_READ>` for the reading side.

The size of the internal fifo of a converter channel `conv` can be set with the method

```
conv.setReadingFIFOSize(int size);
```

The default size is 16. Note that this method has to be called before the converter channel is bound to the fifo reader; otherwise, a runtime error is raised.

### Conversion Semantics

Every time, the SDF writing side writes a token to the converter channel, it gets stored within the internal fifo (possibly after data type conversion). Every time, the fifo reading side performs a read operation, the oldest value within the internal fifo is removed and passed to the reading side. If the internal fifo is empty, the reading side blocks.

### Corner Case Handling

If the internal fifo is full, and the writing side wants to write another token, this situation constitutes a corner case with no obvious semantics. Therefore, the converter channel provides a method

```
setSDF2FIFOfullbuffer(SDF_FIFO_OPT opt);
```

to specify, how this corner case is handled. The possible values for `opt` and the respective behaviour are:

- **error** : a run-time error is raised (default)
- **discardOldest** : the oldest value in the internal fifo is discarded
- **discardCurrent** : the very value to be written is discarded

Which option makes most sense, depends on the application. By setting the error option as a default, it is assured that the designer becomes aware of this corner case, while the error message also gives instructions on the options to choose from.

### 5.1.2 SDF to SC

Here, the converter channel instantiates an internal converter module which is connected to the writing side via an `sca_sdf_signal<T_WRITE>`, and which makes use of the SystemC-AMS `sca_scsdf_out<>` converter port to connect to the reading side via an internal `sc_signal<T>`. There are no options to be set.

#### Conversion Semantics

A token which is written to the internal `sca_sdf_signal<T_WRITE>` by the writing side at SystemC-AMS time  $t$  is converted such that the internal `sc_signal<T_READ>` holds the value of that token from SystemC time  $t$  on. Note that there is a value changed event on the reading side *only* if there is an actual value change of one token to the next one.

#### Corner Case Handling

There are no real corner cases here. However, the designer might be interested in the actual points in time the SDF side is written, although these write events may cause no value change events. Therefore, by using the method

```
conv.clock_sdf();
```

of a converter channel `conv`, a Boolean clock signal can be accessed which runs with the speed of the TDF cluster. That is, every time the converter channel gets a new token from the SDF side, this clock signal will have a rising edge.

### 5.1.3 SDF to ELEC\_VOLTAGE/ELEC\_CURRENT

In this case, the converter channel instantiates an internal voltage or current source, respectively, that is controlled by the SDF side. The choice of which quantity to convert to is set by using the method

```
conv.setElec_Mode(MoC M)
```

of the converter channel `conv`, where `M` is `ahes::ELEC_VOLTAGE` or `ahes::ELEC_CURRENT`, respectively.

If an `sca_elec_port` (e.g. a connector of a capacitor) is bound to the converter channel, it will be actually bound to an internal `sca_elec_node`, which is bound to the *positive* connector of the internal voltage or current source. It is also possible to connect to the respective *negative* connector, which is bound to another internal `sca_elec_node`, which can be accessed with the method

```
conv.neg_elec_node()
```

of a converter channel `conv`. A short code example demonstrates its usage:

```

ahes::converterchannel<ahes::SDF, int> conv;
conv.setElec_Mode(ahes::ELEC_VOLTAGE);
sca_r r1;
r1.p(conv);
r1.n(conv.neg_elec_node());

```

**Listing 5.1:** Using both electrical poles of a converter channel

This connects the resistor `r1` in parallel to the internal SDF-controlled voltage source.

**Note:** If the `neg_elec_node()` method is **not** accessed, the negative connector of the internal voltage/current source will be bound to `gnd`.

## Options

The internal voltage/current source used has an option to scale the voltage/current value which is generated. This factor can be set with the converter channel method

```
conv.setElec_Scaling(double val);
```

of a converter channel `conv`. The default value is 1.

## 5.2 Conversion from FIFO

### 5.2.1 FIFO to SDF

Here, we have a writer which expects to write to a `sc_fifo<T_WRITE>` channel and a SDF reader. Here, the converter channel instantiates an internal `sc_fifo<T_WRITE>` for the writing side and an `sca_sdf_signal<T_READ>` for the reading side. The size of the internal fifo is passed as an argument to the converter channel's constructor .

### Conversion Semantics

A token from the writing side is stored within the internal fifo first. Every time, the SDF side initiates a read (which will happen with a constant pace), the oldest token is taken from the internal fifo, converted to the data type `T_WRITE`, and passed to the SDF side.

### Corner Case Handling

The corner case here is that the internal fifo is empty, and the reading side has to read another token. For this, the converter channel provides a method

```
setFIFO2SDFemptybuffer(SDF_FIFO_OPT opt);
```

to specify, how this corner case is handled. The possible values for `opt` and the respective behaviour are:

- **error** : a run-time error is raised (default)
- **hold** : the last value which could be read from the internal fifo is passed

- **constant** : a constant value is passed. This value is 0 by default (casted to the respective data type), and can be specified explicitly (before the simulation starts) with the method

```
setFIFO2SDFemptybufferConstant(T_READ val).
```

Like in the reverse conversion case, which option makes most sense depends heavily on the application at hand.

## 5.3 Conversion from sc

### 5.3.1 SC to SDF

In this case, the converter channel instantiates an internal converter module which is connected to the writing side by a SystemC-AMS `sca_sc_sdf_in<T_WRITE>` converter port via an internal `sc_signal<T_WRITE>`, and connects to the reading side via an internal `sca_sdf_signal<T_READ>`. There are no options to be set.

#### Conversion Semantics

A value which is written to the internal `sc_signal<T_WRITE>` by the writing side basically constitutes a *current* value for the signal. This current value is then sampled by the SDF side with a frequency determined by its sampling period.

#### Corner Case Handling

There are no *real* corner cases here. However, if the SC side provides value changes with a frequency higher than the sampling frequency of the SDF-side, values will be lost. There is no way to handle this, since the sampling period of an TDF-cluster cannot be changed at runtime. But the converter channel can provide the user with warnings, if this happens. These warnings can be enabled by using the method

```
conv.use_lost_val_alerter();
```

of a converter channel `conv`.

### 5.3.2 SC to FIFO

In this case, we have a SC writer and a reader, which expects to read from a `sc_fifo<>` channel. Here, the converter channel instantiates an internal `sc_signal<T_WRITE>` for the writing side and a `sc_fifo<T_READ>` for the reading side. Apart from that, the conversion semantics are the same as the conversion semantics of SDF to FIFO. The only difference is that the writing side doesn't provide values at a constant pace. For ease of reference, all relevant information is included again in this section.

The size of the internal fifo of a converter channel `conv` can be set with the method

```
conv.setReadingFIFOSize(int size);
```

The default size is 16. Note that this method has to be called before the converter channel is bound to the fifo reader; otherwise, a runtime error is raised.

## Conversion Semantics

Every time, the SC writing side writes a token to the converter channel, it gets stored within the internal fifo (possibly after data type conversion). Every time, the fifo reading side performs a read operation, the oldest value within the internal fifo is removed and passed to the reading side. If the internal fifo is empty, the reading side blocks.

## Corner Case Handling

If the internal fifo is full, and the writing side wants to write another token, this situation constitutes a corner case with no obvious semantics. Therefore, the converter channel provides a method

```
setSC2FIFOfullbuffer(SDF_FIFO_OPT opt);
```

to specify, how this corner case is handled. The possible values for `opt` and the respective behaviour are:

`error` : a run-time error is raised (default)

`discardOldest` : the oldest value in the internal fifo is discarded

`discardCurrent` : the very value to be written is discarded

Which option makes most sense, depends on the application. By setting the error option as a default, it is assured that the designer becomes aware of this corner case, while the error message also gives instructions on the options to choose from.

### 5.3.3 SC to ELEC\_VOLTAGE/ELEC\_CURRENT

In this case, the converter channel instantiates an internal voltage or current source, respectively, that is controlled by the SC side. For the user, the coding for this conversion direction looks almost exactly like SDF to ELEC\_VOLTAGE/ELEC\_CURRENT. For ease of reference, all relevant information is included again in this section.

The choice of which quantity to convert to is set by using the method

```
conv.setElec_Mode(MoC M)
```

of the converter channel `conv`, where `M` is `ahes::ELEC_VOLTAGE` or `ahes::ELEC_CURRENT`, respectively.

If an `sca_elec_port` (e.g. a connector of a capacitor) is bound to the converter channel, it will be actually bound to an internal `sca_elec_node`, which is bound to the *positive* connector of the internal voltage or current source. It is also possible to connect to the respective *negative* connector, which is bound to another internal `sca_elec_node`, which can be accessed with the method

```
conv.neg_elec_node()
```

of a converter channel `conv`. A short code example demonstrates its usage:



```

ahes::converterchannel<ahes::SC, int> conv;
conv.setElec_Mode(ahes::ELEC_VOLTAGE);
sca_r r1;
r1.p(conv);
r1.n(conv.neg_elec_node());

```

**Listing 5.1:** Using both electrical poles of a converter channel

This connects the resistor `r1` in parallel to the internal SDF-controlled voltage source.

**Note:** If the `neg_elec_node()` method is **not** accessed, the negative connector of the internal voltage/current source will be bound to `gnd`.

### Options

The internal voltage/current source used has an option to scale the voltage/current value which is generated. This factor can be set with the converter channel method

```
conv.setElec_Scaling(double val);
```

of a converter channel `conv`. The default value is 1.

## 5.4 Conversion from ELEC\_VOLTAGE/ELEC\_CURRENT

### 5.4.1 ELEC\_VOLTAGE/ELEC\_CURRENT to SDF

In this case, an electrical voltage or an electrical current quantity is measured, respectively, and converted to an SDF signal. Which quantity is measured has to be specified with the MoC template parameter when instantiating the converter channel. For example, `ahes::converterchannel<ELEC_CURRENT, double, int>` measures current.

As in the other conversion direction, the converter channel also has an additional negative `sca_elec_node`, which is bound to the negative connector of the internal measurement unit, and which can be accessed again by using the method `neg_elec_node()`. If it is not accessed, the negative connector of the measurement unit will be bound to `gnd`.

### Options

The internal voltage/current measurement units used have an option to scale the voltage/current value which is measured. This factor can be set with the converter channel method

```
conv.setElec_Scaling(double val);
```

of a converter channel `conv`. The default value is 1.

### 5.4.2 ELEC\_VOLTAGE/ELEC\_CURRENT to SC

In this case, an electrical voltage or an electrical current quantity is measured, respectively, and converted to an SC signal. For the user, the coding for this conversion direction looks almost exactly like `ELEC_VOLTAGE/ELEC_CURRENT` to SDF. For ease of reference, all

relevant information is included again in this section.

Which quantity is measured has to be specified with the MoC template parameter when instantiating the converter channel. For example, `ahes::converterchannel<ELEC_CURRENT, double, int>` measures current.

As in the other conversion direction, the converter channel also has an additional negative `sca_elec_node`, which is bound to the negative connector of the internal measurement unit, and which can be accessed again by using the method `neg_elec_node()`. If it is not accessed, the negative connector of the measurement unit will be bound to `gnd`.

### Options

The internal voltage/current measurement units used have an option to scale the voltage/current value which is measured. This factor can be set with the converter channel method

```
conv.setElec_Scaling(double val);
```

of a converter channel `conv`. The default value is 1.

### 5.4.3 ELEC\_VOLTAGE/ELEC\_CURRENT to FIFO

In this case, an electrical voltage or an electrical current quantity is measured, respectively, and written to an `sc_fifo`. Internally, there is an `ELEC_X` to `SC` conversion first, followed by a `SC` to `FIFO` conversion. For the user, the coding for this conversion direction looks almost exactly like `ELEC_VOLTAGE/ELEC_CURRENT` to `SC` as well as `SC` to `FIFO`. For ease of reference, all relevant information is included again in this section.

Which quantity is measured has to be specified with the MoC template parameter when instantiating the converter channel. For example, `ahes::converterchannel<ELEC_CURRENT, double, int>` measures current.

As in the other conversion direction, the converter channel also has an additional negative `sca_elec_node`, which is bound to the negative connector of the internal measurement unit, and which can be accessed again by using the method `neg_elec_node()`. If it is not accessed, the negative connector of the measurement unit will be bound to `gnd`.

### Options

The internal voltage/current measurement units used have an option to scale the voltage/current value which is measured. This factor can be set with the converter channel method

```
conv.setElec_Scaling(double val);
```

of a converter channel `conv`. The default value is 1.

The size of the internal fifo of a converter channel `conv` can be set with the method

```
conv.setReadingFIFOSize(int size);
```

The default size is 16. Note that this method has to be called before the converter channel is bound to the fifo reader; otherwise, a runtime error is raised.

## Corner Case Handling

If the internal fifo is full, and the writing side wants to write another token, this situation constitutes a corner case with no obvious semantics. Therefore, the converter channel provides a method

```
setELEC2FIFOfullbuffer(SDF_FIFO_OPT opt);
```

to specify, how this corner case is handled. The possible values for `opt` and the respective behaviour are:

- **error** : a run-time error is raised (default)
- **discardOldest** : the oldest value in the internal fifo is discarded
- **discardCurrent** : the very value to be written is discarded

Which option makes most sense, depends on the application. By setting the error option as a default, it is assured that the designer becomes aware of this corner case, while the error message also gives instructions on the options to choose from.

## 6 Data type conversion

The intention of the data type conversion capabilities of the converter channels is to provide obvious and simple data type conversion semantics, since designers might feel uncomfortable with data type conversion hidden within a channel. In fact, there were explicit doubts expressed by our industry partners on this matter. However, by leaving the template parameters `T_READ1` and `T_READ2` unspecified, the data type conversion capabilities of a converter channel can be effectively turned off.

In general, there are two mayor issues when converting values from one data type to another:

- When converting decimal numbers (like `double` values) to integer numbers (like `int`), there is the issue how to round the original number.
- If a data type has a larger value range than the target data type, possible overflows have to be handled

For the first issue, the converter channel offers an option, which can be accessed by the method

```
float2IntPolicy(ahes::Float2IntPol pol);
```

where `ahes::Float2IntPol` is an enum type which can take the values `ROUND`, `FLOOR` and `CEIL` for rounding by value, rounding down, and rounding up, respectively.

For the second issue, there is a converter channel method

```
overflowClipping();
```

which causes all values exceeding the value range of the target data type to be clipped to the respective maximum or minimum value the target data type can hold. If this option is not set, the data type conversion will behave in a modular arithmetic manner. I.e., the target value will

be the remainder of the original value when divided by the target data type's maximum.

Another issue is application specific: In mixed signal systems, there are situations where values from a continuous domain have to be quantized, i.e. mapped to a value with a fixed value range. The most prominent example is an A/D converter. Here, the values are not simply converted by using some rounding approach, but effectively scaled and rounded. For example, the value range on the analogue side may be an interval  $[-10,10]$ , which is mapped to 5-bit numbers, whose natural value range is  $[0,31]$ . Using rounding here makes no sense, since the negative values would be discarded, while most of the target range would not be reached.

To address this issue, an options has been included which leads to the scaling of a certain value range of the writing side data type to the value range of the reading side data type. The option can be used by calling the method

```
conv.setRangeScaling<T_READ>(T_WRITE min, T_WRITE max);
```

of a converter channel `conv`. `T_READ` has to be one of the data types `sc_bv<N>`, `sc_lv<N>`, `sc_uint<N>`, `sc_int<N>`, `sc_biguint<N>` or `sc_bigint<N>`. `T_WRITE` must be an ordinary C++ numerical data type (e.g. `int` or `double`), or a SystemC fixed-point data type like `sc_fixed<...>`. The minimum and maximum on the *reading* side is determined by  $n$ , namely  $-2^{n-1}+2$  resp.  $2^{n-1}-1$  for `sc_int<N>`. This determines a scaling factor  $fac := (2^{N-1} - (-2^{N-1}+2)) / (max - min)$  and a scaling function  $f(x) := (x - min) * fac + (-2^{N-1}+2)$ . That is, the interval passed to the method `setRangeScaling` is mapped linearly to the value range of the reading side data type, effectively implementing uniform quantization. If `setRangeScaling` is called with a `T_WRITE`, `T_READ` pair which does not support scaling, a compile time error is raised for better debuggability. An exception is, if `T_READ` is `double`, this is a case which can only be handled at runtime. If `T_READ` does not represent a type, which is used in the converter channel a runtime error is raised at the begin of the simulation.

Figure 6.1 shows an overview of the data type conversion capabilities of the converter channels. It is shown for each `T_WRITE`, `T_READ` pair, how the conversion is handled, and what options are possible to use. In total, we have 11 different cases, and each case is explained briefly below the table. Below the icon for each case in the explanation, it is indicated which of the three options mentioned above can be used. The letter **C** indicates that the clipping-option can be used, the letter **R** indicates that the rounding option can be set, and the letter **S** indicates that value range scaling can be used. In any case, the user will get warnings before and during the simulation on any problematic data conversion issue, e.g. when there is an `T_WRITE` value during simulation which does not lie within the previously defined value range scaling interval, or before the simulation when the `T_WRITE` range exceeds the `T_READ` range, e.g. in the case of converting from `double` to `int`.

Here are some miscellaneous notes on the converter channels data type conversion facilities:

- Conversion *from* `sc_logic` and `sc_lv<N>` is not supported (except to `sc_lv<N>`), since there are no obvious ways to convert the values `sc_dt::Log_X` (undefined) and `sc_dt::Log_Z` (high impedance) to numerical values.
- The data type `long double` is not supported since using this data type causes ambiguity and overload problems within the core System-C 2.2.0 library on the x86 platform with the C++ compilers used for developing the converter channels (GNU (GNU Not Unix) C++ 4.1 and 4.3.2).

Modelling extensions for polymorphic signals, final library elements

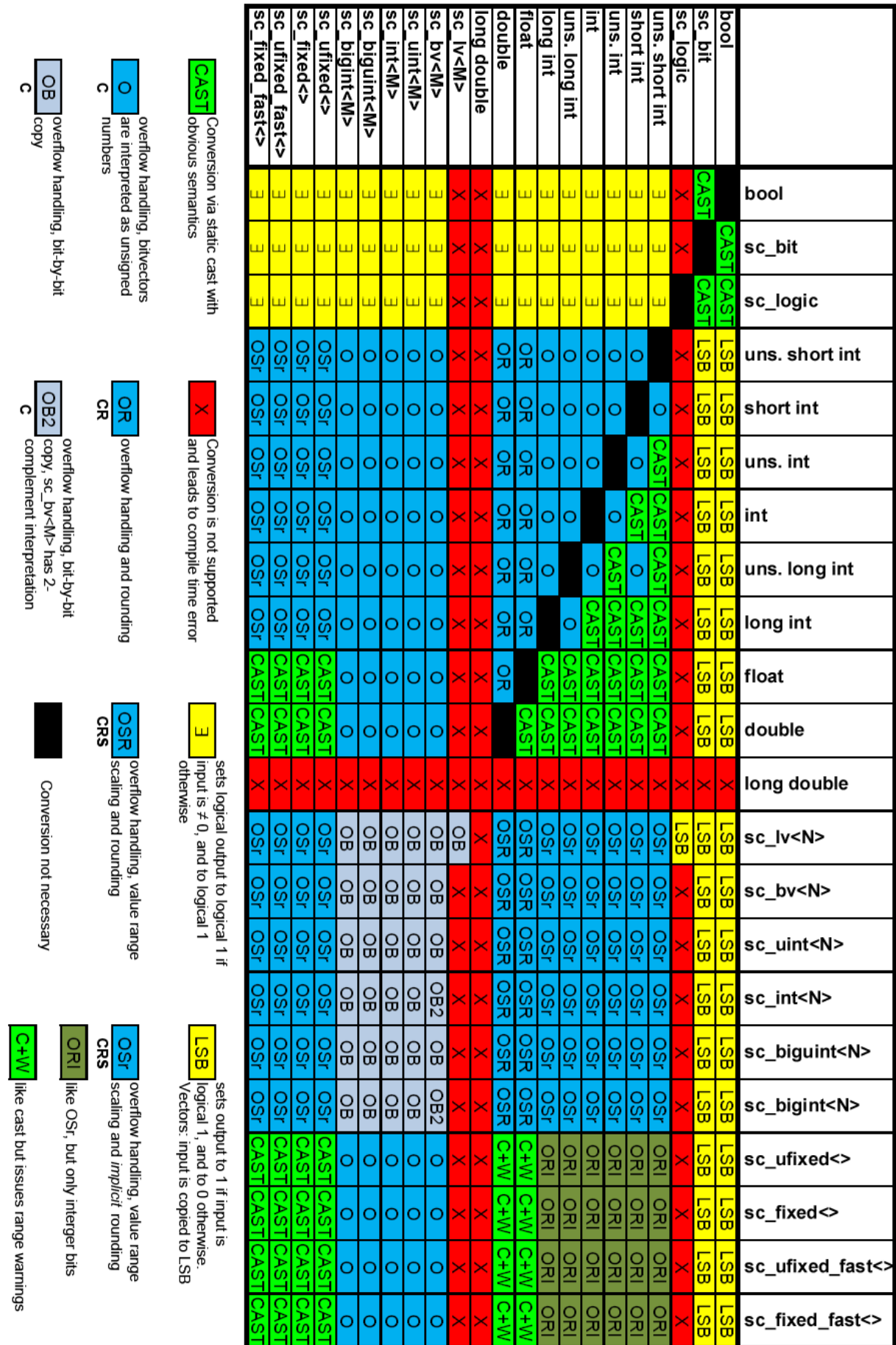


Figure 6.1 The converter channel's data type conversion capabilities

- When converting a logical type (`bool` and `sc_bit`) to a numerical value, a logical 1 is converted to a numerical 1, and a logical 0 is converted to a numerical 0. In the case of bit-vectors, the logical value will be copied to the least significant bit (LSB).
- When using value range scaling, the rounding option refers to the *scaled* values. Therefore, even when converting from e.g. `int` to `sc_int<N>`, setting the rounding option becomes meaningful, when value range scaling is used.
- Conversions from an N bit bit-vector to an M bit `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint`, vector, with  $N > M$  ignores the higher bits of the vector and just uses the lower M bits if the clipping option is not set. If clipping is set the maximum possible value is set.
- Conversions from integer or bit-vector types to `sc_fixed`, `sc_ufixed` or the fast versions of those types sets just the integer part for the fixed point numbers. An N bit integer value can be lossless converted to an N integer part fixed point number.
- In conversions from fixed point numbers to the System-C integer types, the integer types need one bit more of width compared to the integer part of the fixed point number, since the value range is by one bit larger, through rounding of the fractional positions. So an N bit integer part fixed point number needs an N+1 bit integer or vector as target.
- The conversion between various specializations of the System-C fixed point numbers is done by an ordinary cast, to facilitate the full spectrum of quantisation and rounding options provided by those data types.
- According to tests the memory consumption during compilation increases non linearly with an order greater than  $O(n)$ , where n denotes the number of pairwise different data type conversions. Therefore it is recommended, to split large numbers of converter channels up into different compilation units and linking them together afterwards. Usage of large numbers of converter channels with the same conversion pairs within the same compilation unit does not cause the compiler to use excessive memory. The large memory usage is a design decision and is a trade off for increased simulation and runtime performance.
- Mistakes on the user side, which where runtime errors in earlier versions, eg. enabling scaling for conversions which do not support scaling or tried conversation from `sc_lv` or `sc_logic` to numbers now cause compile time errors for better feedback and ease of debugging ones modules using converter channels.
- Conversions with enabled scaling from unsigned data types to signed types are only able to scale to values within the positive range of the target type.
- Value range scaling, if enabled, is done within the value range of the writing data type, therefore, you cannot scale up to the full value range on the reading side, if that data type has a larger width in bits on the reading side of the channel.
- For a conversion from `sc_fixed`, `sc_ufixed`, `sc_fixed_fast` or `sc_ufixed_fast` to `bool`, `sc_bit` or `sc_logic`, the “fixed” value has to be exactly 0 to result in false or 0. There is no confidence interval.

- Conversions from/to fixed point values are implemented, but are not recommended for usage, since the semantic is not universally clearly defined and can be interpreted differently by different people. It is consistent within the converter channels though. The user should consider carefully, if our semantic fits his purpose exactly.

## 7 Current distribution

The distribution of the converter channels consists of 3 files, `converterchannels.h`, `converters.h` and `converters_data.h`. Those files implement the C++ template based functionality. `Converterchannels.h` contains the only class needed by the user, it is the toplevel view of the converter channels. In `converters.h` the internally used classes, needed for the conversion between different MoCs (models of computation), are implemented. In `converters_data.h` all classes are located which handle the conversion between different data types.

The converter channels additionally need the `ahes_helpers` library, which implements a wrapper around the GNU (GNU Not Unix) C++ compilers name demangling function, to provide easily readable type information in warning or information messages. For other compilers, or gcc versions, not having the needed core function, the library contains a stub which just returns the mangled name.

## 8 Conclusion and future work

This report presented the final status of the converter channels for the ANDRES project. Compared to the Deliverable 1.5a [2], there have been considerable improvements:

- The support for electrical networks was included, together with all respective conversion directions, with the exception of FIFO to ELEC\_X.
- The conversion direction SC to FIFO has been included
- SC to SDF undersampling detection has been implemented
- A clock signal can be generated with the pace of the SDF writing side.
- The data type conversion capabilities have been reworked completely. The data type conversions performed by the converter channels are as intuitive and transparent as possible, while discarding conversions with no obvious conversion semantics (like conversion from `sc_logic`) and platform-dependant ambiguities (like conversions involving `long double`).
- The general structure regarding the coding of the converter channels has been reworked.

Future work will involve a decision on FIFO to X conversion (with X different from SDF). A possible conversion semantic for these cases has been sketched at the beginning of section 5, and can be implemented quickly. Yet, it is unclear if this makes sense with regard to reasonable applications, such that these conversion directions have to be discarded altogether. Apart from that, further testing will be done, also involving porting & testing with Microsoft compilers, and the development of more usage examples.

## 9 References

- [1] Rüdiger Schroll. *Design komplexer heterogener Systeme mit Polymorphen Signalen*. Dissertation, Institut für Informatik, Universität Frankfurt am Main, 2007
- [2] ANDRES Deliverable D1.5a
- [3] A. Herrholz, F. Oppenheimer, A. Schallenberg, W. Nebel, C. Grimm, M. Damm, F. Herrera, E. Villar, A-M. Fouilliant, M. Martínez "ANDRES- ANalysis and Design of run-time REconfigurable, heterogeneous Systems" Workshop on "Adaptive Heterogeneous Systems-On-Chip and European Dimensions" in the Design Automation and Test in Europe 2007, DATE'07, Nice, France. DATE 07 Friday Workshop Notes pp. 64-71.
- [4] C. Grimm, R. Schroll, K. Waldschmidt, F. Brame. *Mixed-Level Simulation heterogener Systeme*. In: Multi Nature Systems, S. 5, Erfurt, 2007.