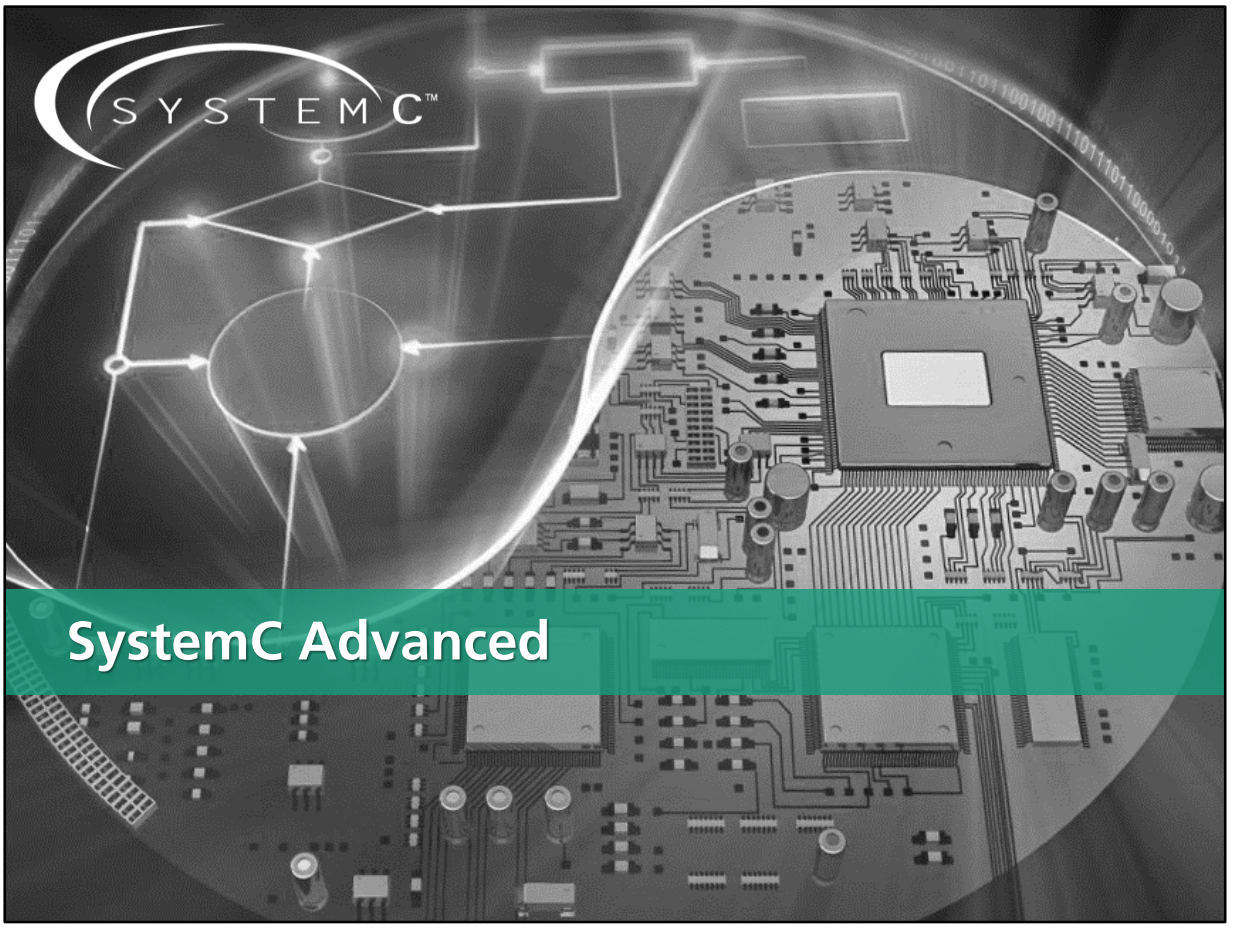
SystemC and Virtual Prototyping

Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

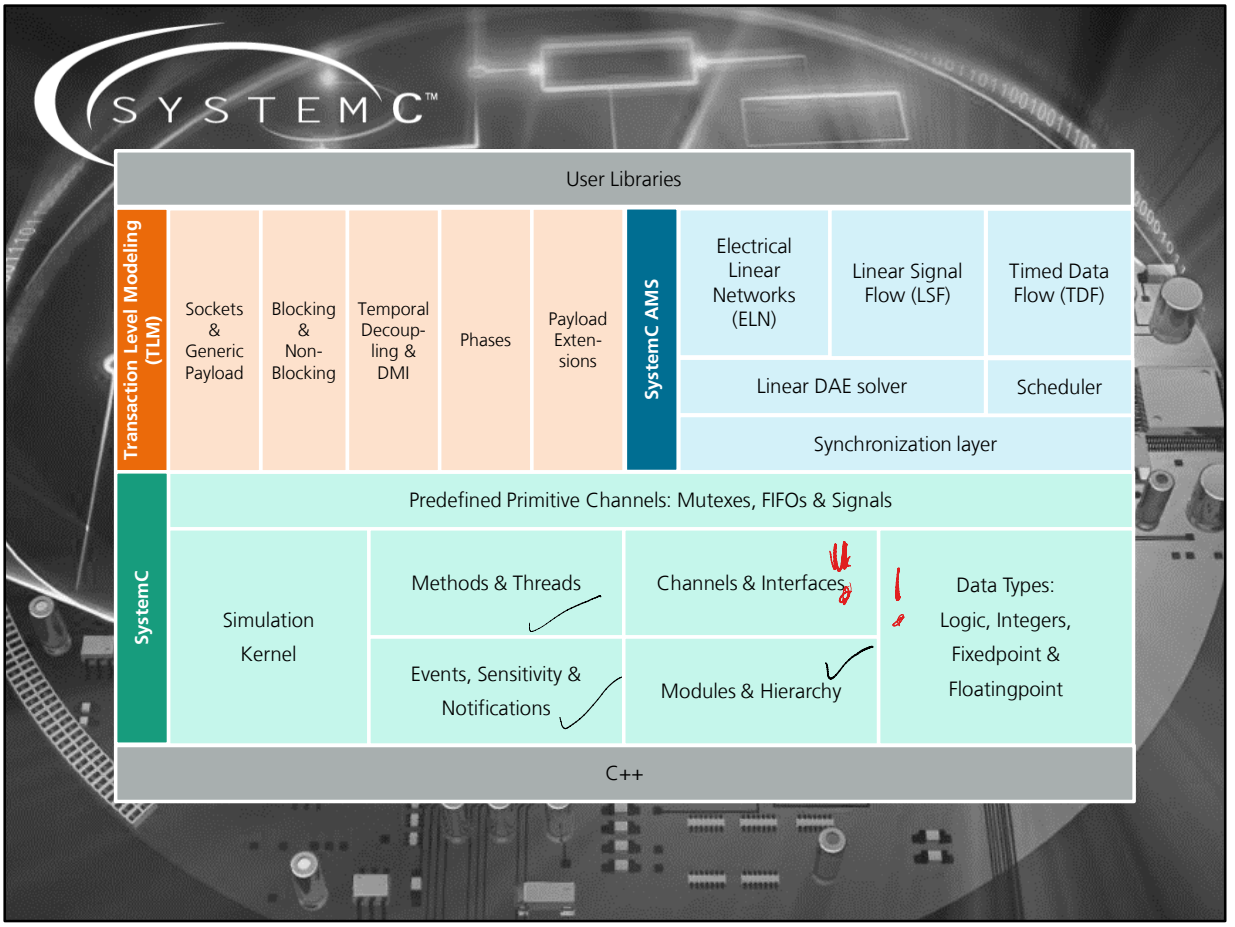
Fraunhofer IESE

Your notes:



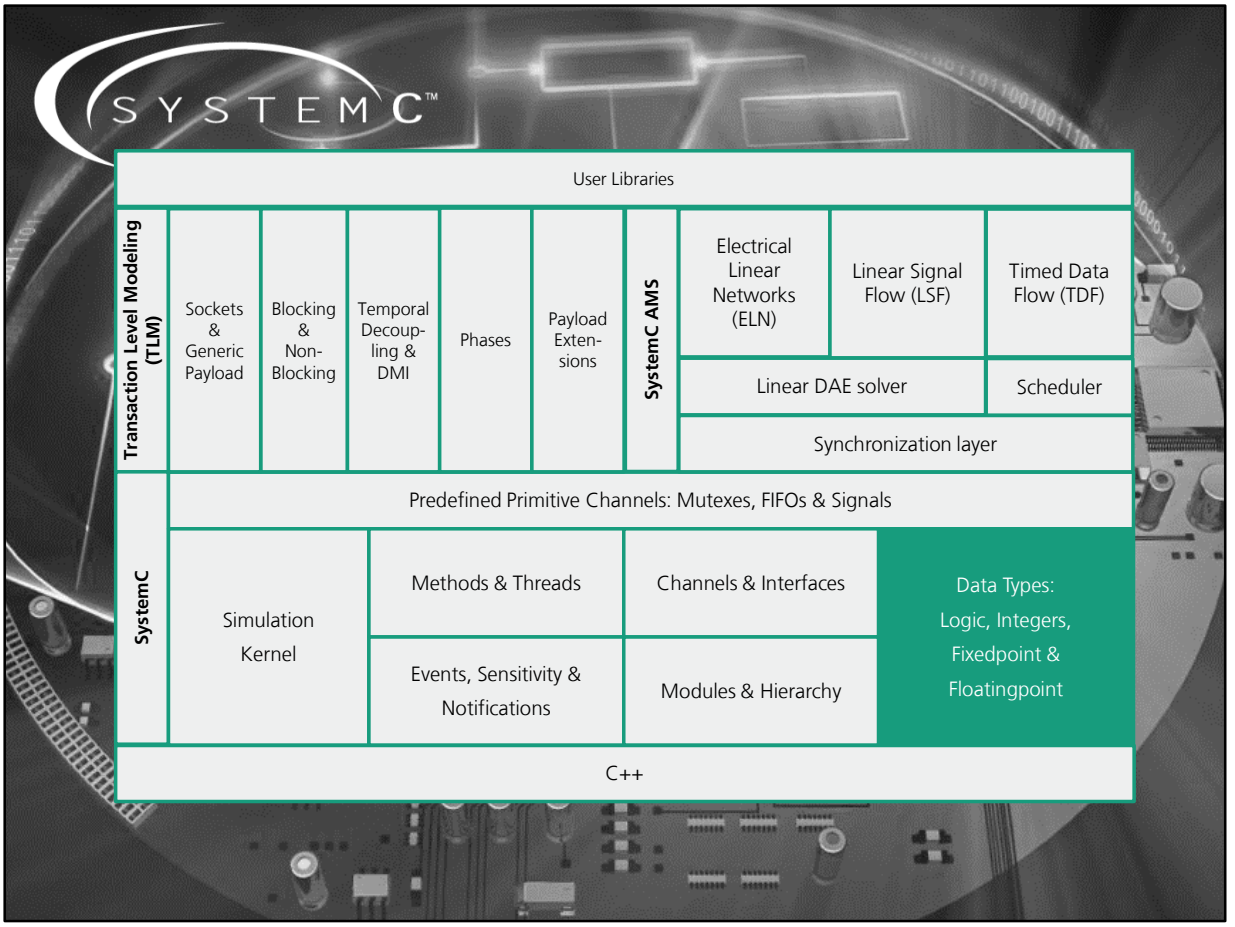
Your notes:

we talked about Systemc basics, how we can build a component, how we can describe behavior, we have seen that we have methods and threads, this are the 2 ways to describe the behavior of components, we have talked about tracing, where we can trace the delta cycles, But not as Professional in synopsys



Your notes:

(1) you know System-C is based on C++, that means that we inherit all the data types that C++ offer.



Your notes:

C++ Datatypes

Data Type	Size [Bit]	Range
`bool`	1	0 to 1 (true, false)
`char`	8	0 to 255
`short int`	16	-32,768 to 32,767
`unsigned short int`	16	0 to 65,535
`int`	32	-2,147,483,648 to 2,147,483,647
`unsigned int`	32	0 to 4,294,967,295
`long long int`	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`	64	0 to 18,446,744,073,709,551,616
`float`	32	-3.4E+38 to +3.4E+38
`double`		-1.7E+308 to +1.7E+308

Fraunhofer IESE

Your notes:

(1) We have here a list of Datatypes

(2) You can use this data Types for your variables and your signals.

SystemC Logic Datatypes

- In C++ there is the datatype `bool` with values `true` and `false`
- For hardware modeling this is not enough
- For example: VHDL's `std_logic` (9 States, Verilog has even 12):
  - `U`: Uninitialized
  - `X`: Unknown
  - `0`: 0
  - `1`: 1
  - `Z`: High Impedance
  - `W`: Weak Unknown
  - `L`: Weak 0
  - `H`: Weak 1
  - `-`: Don't Care

more states

(2) For eXs This Logic Data Types.

(3) There was ~~a data type~~ For single Bit but here you can use Bool data Type with values True False

Fraunhofer IESE

Your notes:

(1) Additionaly SystemC Prouides Custom Data types.

(4) But for H/W modeling it is not enough when we look to VHDL we have std_logic and it has more states as shown above, not true or False only.

(5) So the H/W designers they are used for this States it makes sense From H/W Point of view to have such states.

what a SystemC doing regarding this?
Here is 697 and bit vector we have just to 81

SystemC Logic Datatypes: sc_bit, sc_bv<W>

SC_vector

```
sc_bv<2> a = 2;
sc_bv<2> b = "10";
std::cout << a << std::endl; // 10
a = 5;
std::cout << a << std::endl; // 01 overflow
a = a | b;
std::cout << a << std::endl; // 11
bool c = a.and_reduce();
std::cout << c << std::endl; // 1
sc_bv<6> d = "000000";
d.range(0,3) = "1111";
std::cout << d << std::endl; // 001111
std::cout << d(0,3)<< std::endl; // 001111
std::cout << d.range(0,3) << std::endl; // 001111
std::cout << d[0] << std::endl; // 1
d = (a, d.range(0,3));
std::cout << d << std::endl; // 111111
```

2 bits
you can Assign Decimal or string
if you Assign Bigger Number
it make bit wise And operation with all the elements then it return 1 bit
if you create something like this for ex & want from bit 0 to 3 with 1's
concatination
3 ways of writing range

1. sc_bit: deprecated, use bool instead!
2. sc_bv<W>: bit vector
   - Width as template parameter
   - Typical operators overloaded: &, |, ^, ~, …
   - X_reduce() methods
   - Ranges
   - Concatenation
   - Similar VHDL's bit_vector

we have one to show single Bit.
if you want to have variable bits.
it's reducing Algorithms.

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/datatypes

here it's explanation for features with SystemC Logic Data Types

Fraunhofer IESE

Your notes:

systemc logic vector

SystemC Logic Datatypes: sc_logic, sc_lv<W>

Here we have SC_Logic where we have more than 2 states.

- sc_logic features 4 States:
  1. 'X': Unknown
  2. '0': 0
  3. '1': 1
  4. 'Z': High Impedance
- sc_lv<W> vector of sc_logic
  - Similar to sc_bv<W>
  - Special case tristate bus systems: if several processes want to drive the same signal special signal classes sc_signal_resolved and sc_signal_rv<W> have to be used.

"Logic vector"
"it's a vector with all this states"

we could need this for e.g.

Fraunhofer IESE

Your notes:

Remember: Fixed Point and Two's Complement Numbers

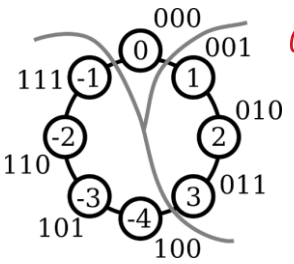
- Positive Fixed point:

$$< \underbrace{d_{n-1}\, d_{n-2} \ldots d_1\, d_0}_{n \text{ digits left}} . \underbrace{d_{-1} \ldots d_k}_{k \text{ digits right}} > = \sum_{i=-k}^{n-1} d_i \cdot 2^i$$

$\pi = 000011.001001_2 = 3.140625_{10}$

- Two's Complement: we have the Advantage to not have double values for 0

$$< d_{n-1} \ldots d_0 . d_{-1} \ldots d_k > = \left(\sum_{i=-k}^{n-2} d_i \cdot 2^i \right) - d_{n-1} \cdot (2^{n-1})$$



Advantage of 2's compliment

1. No double 0
2. Asymmetric range
3. Simple hardware performing (add, sub …)

Fraunhofer IESE

Your notes:

→ In SystemC we have also the Possibility of Fixed Points and 2's Compliment things

SystemC Integer Datatypes: sc_int<W>, sc_uint<W>, …

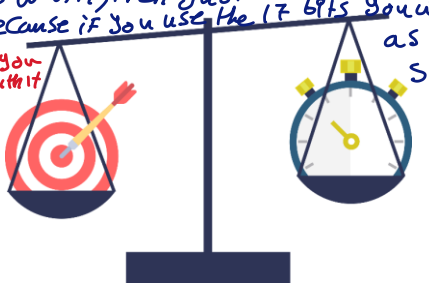
we have ① int 2's compliment signed ② int 2's compliment unsigned

we have ①

- sc_int<W> for signed integers and sc_uint<W> for unsigned integers
  - Provides efficient way to model data with specific widths (1-64) you can specify the width with this range
  - When modeling numbers where data width is not an integral multiple of the simulating processor's data paths, some bit masking and shifting must be performed, which leads to an overhead in wall clock time.

if it's not possible to model it with 17 bits width, Then just use 64 bits integer and not the systemC int with 64 bits because if you use the 17 bits you will have overhead as systemC do something more

But if you have no alternative due to Quality so you go with it



- sc_bigint<W> and sc_ubigint<W>
  - Support lage data width (e.g. 512)
  - Cost of speed!

This when you need huge data width, but remember that your system is dealing with 64 bits but systemC do some stuff to fix this difference but it affects the speed.

Fraunhofer IESE

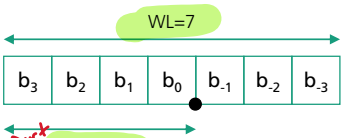
Your notes:

SystemC Fixpoint Datatypes: sc_fixed<…>, …

There is two ways to do it: ① Template ② Dynamically

- sc_ufixed<WL, IWL, [QMODE], [OMODE]> a;
- sc_ufix a(WL, IWL, [QMODE], [OMODE]);
- sc_fixed<WL, IWL, [QMODE], [OMODE]> a;
- sc_fix a(WL, IWL, [QMODE], [OMODE]);

- Example: sc_ufixed<7,4>:

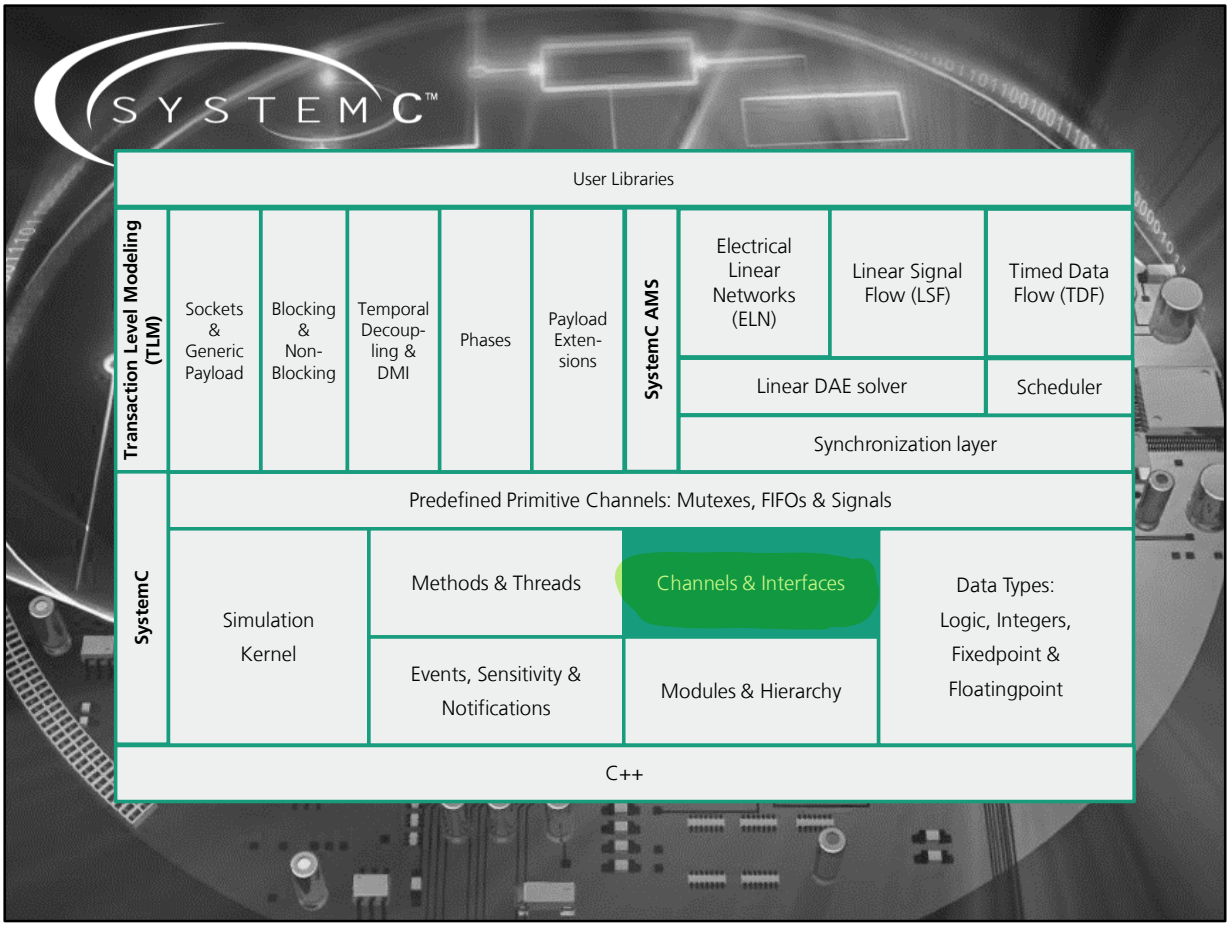


WL = Word Length
IWL = Integer WL

- QMODE: Quantization Mode: SC_RND, SC_TRN …
- OMODE: Overflow Mode: SC_WRAP, SC_SAT …
- See SystemC Standard for more details!

Fraunhofer IESE

Your notes:



Your notes: if you understand that you will be understanding how systemc really works.

Recall: Polymorphism – Pure Virtual (Abstract Base Classes)

```cpp
#include <iostream>
using namespace std;

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      virtual void area() = 0;
};
```

```cpp
[ ... ]

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,5);
   Triangle  tri(10,5);

   shape = &rec;
   shape->area();
   shape = &tri;
   shape->area();

   return 0;
}
```

Output:
Rectangle class area: 50
Triangle class area: 25

- Only pointes to abstract classes can be created, no objects!
- Child classes must implement virtual function! Otherwise compiler crashes!
- Why using it? For structuring! For defining Interfaces

Fraunhofer IESE

Recap

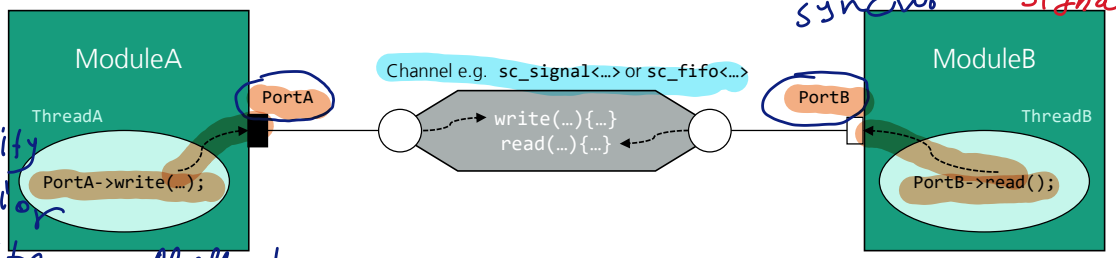
A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract. Classes containing pure virtual methods are termed "abstract" and they cannot be instantiated directly.

An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that operate similarly. The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application. This architecture also allows new applications to be added to a system easily, even after the system has been defined.

Your notes:

Closer Look on Ports, Signals, Interfaces and Channels

- VHDL and Verilog use signals for communication
- In SystemC a signal (i.e. `sc_signal`) is just a special case of a *Channel*
- *Channels* separate communication from functionality
- *Channels* are containers for communication protocols and sync. events
- An *Interface* defines a set of pure virtual methods
- *Channels* implement one or more *Interface(s)*
- *Modules* access *Channel's Interfaces* via bounded *Ports*



Fraunhofer IESE

The interface to be given to the port as the first template parameter is an abstract base class. Essentially, this class consists of the methods that can be called inside the module on the port. However, these are purely virtual methods, and therefore not implemented in the interface class. By passing the interface class to the port, only the method heads are known, such that in the module, which instantiates the port, can call the methods. It is important to know that the port itself does not implement these methods! The methods are implemented in the channel, and binding the channel to the port effectively executes the methods of the channel when a method call occurs in the module. A port therefore forwards the calls of the interface methods in the module to the channel that was bound to the port.

Your notes:

for exp sc_signal there is a specific interface and the interface describes basically which fn's are there to communicate with this channel, and in case of systemc signal these fn's are "write" & "read", This specified in the interface class and then the systemc signal class is derived from this and has to implement logic for performing

a write to signal or a read from a signal.

∘∘ specify Modules Port Just with the Interface, Then the module Knows when I want to Communicate over this Port I will call a write fn and if I want to read I will call read fn, But the Module doesn't know which channel is there, is it signal or FIFO or Mutex and This really nice concept because you can then exchange a signal with a FIFO or with a Buffer, that's the idea because the Interface of all these signals & FIFO's is the same you can just Interchange Communication Models, So you can & Put in ethernet network or CANBus or SPI Bus, that's the idea your modules don't know which channel is there because they all use same Interface for write & read and then you can Interchange easily.

مهم

ال Interface هو بيتكون من Pure virtual methods يعني fn's أقدر أستخدمها

فال Communication ما بين ال modules فأهنا ال Channel هي ال Communication Protocol

وهي اللي بتنفذ أو تـ call ال methods بتاعت ال Interface عشان تتنفذ.

awrite to signal or read From a signal -
so Channels implement 1 or more Interfaces, an

Interface: Example sc_signal

```
template <class T>
class sc_signal_in_if : virtual public sc_interface {
    ...
        virtual const T& read() const = 0;
    ...
};

template <class T>
class sc_signal_write_if : virtual public sc_interface {
    ...
        virtual void write(const T&) = 0;
    ...
};

template <class T>
class sc_signal_inout_if : virtual public sc_signal_in_if<T>, public sc_signal_write_if<T> {
    ...
};

template <class T>
class sc_signal: public sc_signal_inout_if<T>, public sc_prim_channel {
    ...
    T& read() { ... }

    void write(const T&) { ... }
    ...
}
```

Interfaces are a collection of pure function methods

Interfaces can be composed also by inheritance

Channels implement the virtual functions specified by the interface

Fraunhofer IESE

Your notes:

Interface: Example sc_signal

```
class Module : public sc_module {
  ...
  sc_port< sc_signal_in_if<int> > Foo;
  sc_port< sc_signal_inout_if<bool> > Bar;
  ...
  sc_in<int> foo;
  sc_out<bool> bar;
  ...
  // General port declaration:
  sc_port< Interface, N, Policy >
  ...
  Bar->write(10);
  bar.write(10);
}
```

Easier and more convenient to use, especially for RTL modelling

Calling Interface methods with . for specialized ports and -> with standard ports

- Specialized ports `sc_in`, `sc_out`, `sc_inout` for `sc_signal`, for RTL modelling and easy use.
- sc_port has several parameters:
  - Interface (required)
  - N (optional): max number of channels to be bound
  - Policy (optional):
    - SC_ONE_OR_MORE_BOUND
    - SC_ZERO_OR_MORE_BOUND
    - SC_ALL_BOUND
- Binding Errors

Fraunhofer IESE

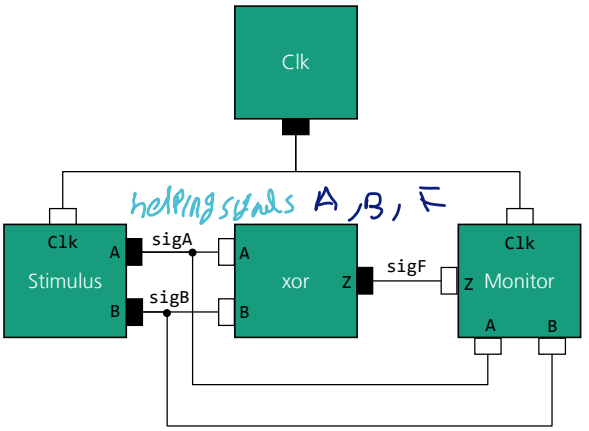
- There exist several binding policies for sc_port:
  - SC_ONE_OR_MORE_BOUND: Is the default value. The port can be bound to 1 to N channels. This implies that the port must also be bound to at least one channel.
  - SC_ZERO_OR_MORE_BOUND: The port can be bound to 0 to N channels. This implies that the port must not be bound.
  - SC_ALL_BOUND: Exactly N channels must be bound.
- If one of these policies is hurt, binding errors will occur, i.e. an error message is displayed during the program's elaboration phase that indicates a missing binding.

Your notes:

Recap: Connecting Modules (Binding)

```
int sc_main(int argc, char* argv[]) {
    sc_signal<bool> sigA, sigB, sigF;

    sc_clock clock("Clk", 10, SC_NS, 0.5);

    stim Stim1("Stimulus");
    Stim1.A.bind(sigA);
    Stim1.B.bind(sigB);
    Stim1.Clk.bind(clock);

    exor2 DUT("xor");
    DUT.A(sigA);
    DUT.B(sigB);
    DUT.Z(sigF);

    Monitor mon("Monitor");
    mon.A(sigA);
    mon.B(sigB);
    mon.Z(sigF);
    mon.Clk(clock);

    sc_start();  // run forever

    return 0;
}
```

- The methods are implemented in the channel
- Binding the channel to the port during runtime: A port forwards the calls of the interface methods in the module to the channel that was bound to the port.



Fraunhofer IESE

Your notes:

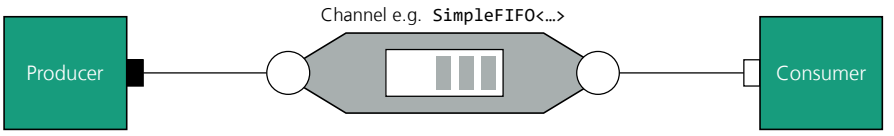
⇒ yesterday we saw if we have wrapper around this and we connect an oP Port with another oP Port from the wrapper we dont need the oP Signal.

⇒ In this example the Channel is the signal, every time we call for ex A.write, the write will be forwarded to the write Fuction which is implemented in the signal class and the Port doesnt know which channel it's just knows my channel should have read Fn, my Channel should have write fn Thats we can easily interchange a signal with a FIFO for ex, because they have same interface.

SimpleFIFO: A Custom Channel Example

systemC offers FIFO class, but to explain you how the concept way i think it's the best way if we develop our custom channel which is a FIFO

- SystemC allows the creation of custom channels according to your needs
- Interface methods are allowed to block by calling wait statements
  (Note that only in SC_THREADs these methods can be called)



- `SimpleFIFO` should implement blocking read and blocking write
- `SimpleFIFOInterface` should have pure virtual functions for read and write

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/custom_fifo

Fraunhofer IESE

Your notes:

(4) So system-C allows the creation of custom channels according to your needs, so for ex, if you want to have a FIFO that has just 5 elements or something like that, we can model that easily by doing own thing, but of course system C has something like this, so we don't need to do this but here i want to really do this because i want you to understand the concept of a channel

② So what should our FIFO do, so interface method are allowed to block by calling wait statements, This gives already kinda of Implication, You remember that systemC method is not allowed to call wait statement,

③ So if our producer here is producing some data in a systemC method and then write to the Channel and inside the channel await function is called then we will get an error. so that implies when we use our FIFO we can only do this with SystemC threads otherwise it wont work and now you see why we need systemC threads in order to model this kind of very sophisticated Channels

④ So our simple FIFO should Implement a blocking read and blocking write, so the consumer doesn't execute further until there is something again in our FIFO, Blocking read means that the consumer can't execute further until something is in it and blocking write means that the producer can't produce anything new if the FIFO is full.

⑤ our FIFO Interface that we build should have pure virtual Functions for read and write

6 So that's what we want to develop

```
#include <iostream>
#include <systemc.h>
#include <queue>
using namespace std;
```

→ we take from stl library because we don't want to program FIFO by our self we just use the queue (in C++) that we can build FIFO

class simpleFIFO

↳ we create simple FIFO Interface and we drive this from sc_interface we have to do this otherwise systemc doesn't recognite this as FIFO Interface.

- Then we have public things, we want to have write fn & read fn.

- we can do that we want to have FIFO only for Integers but we will keep it generic so we use template.

Template

```
template<class T>
```

- we want to have pure virtual and set it to 0.

- Thats our Interface, The Interface describes now how we can comunicate with a FIFO.

Now we will create the FIFo since we drive FIFo From the Interface, also the FIFo Class itself have to be Templated.

Channel

```
template < class T >
class simpleFIFo : public
                   simpleInterface
```

- we derive it From the simple FIFo Interface which is also templated and thats what we call channel.

- We need to add the mems which is also getting elements of Type T which we call FIFo and this should be private so only our class itself it's allowed to access the actual FIFo where we store the Actual Data.

- Then we will work a little bit with events, we will have sc_event which will be basically notified when something is written. "written event", another event which will be the "read event", so we will need this later on in order to unblock or unlock the consumer & Producer.

- Then we will define the max size for our queue and then the PublicStreFFcones.

- We need our Constructor "simple FIFo" and we said our default size of our queue is 16, Then we store this in max size, That's the initialization.

- we couldn't compile because there are 2 fn's missing "read" & "write"

- T read() { we need to implement this blocking behavior if we read from this and the thing is empty.

- if fifo is empty wait until something is written.

- we take the first element of the Fifo and then Pop it out.

- Then we have a read event and notify immediately because this read event is there to block with await statement the write process when the thing is Full and somebody has read from the queue there is a new space so we can continue writing in it, Then we will return the value That's the read method, This really the Fifo behavior if it's empty we have to wait and if have something in it then we take the Front element and read it in our variable and remove it there and notify the read event and return.

* Then we defines the Producer and Consumer. The interesting part will happen, The Producer will have a Port and the Port is described by an Interface "simple FIFO Interface" and lets say we have Integers here and we will call this Port "master"

* Then CTOR is the constructor, it should have the same name as the class itself "Producer" and now as we mentioned we need a thread, so we will have a process that will Produce something and this will be our SystemC Thread.

* and what should the Producer do in the Process, wait for a sec and then we take the master Port and we call now the write function

SimpleFIFO: A Custom Channel Example

```
#include <iostream>
#include <systemc.h>
#include <queue>

using namespace std;

template <class T>
class SimpleFIFOInterface : public sc_interface
{
    public:
    virtual T read() = 0;
    virtual void write(T) = 0;
};
```

Create an Interface for our SimpleFIFO Channel

The FIFO will be accessed by simple read and write methods

Fraunhofer IESE

Your notes:

SimpleFIFO: A Custom Channel Example

```
template <class T>
class SimpleFIFO : public SimpleFIFOInterface<T> {

  private:
    std::queue<T> fifo;
    sc_event writtenEvent;
    sc_event readEvent;
    unsigned int maxSize;

  public:
    SimpleFIFO(unsigned int size=16) : maxSize(size) {}

    T read() {
        if(fifo.empty() == true) {
            wait(writtenEvent);
        }
        T val = fifo.front();
        fifo.pop();
        readEvent.notify(SC_ZERO_TIME);
        return val;
    }

    void write(T d) {
        if(fifo.size() == maxSize) {
            wait(readEvent);
        }
        fifo.push(d);
        writtenEvent.notify(SC_ZERO_TIME);
    }

};
```

Create the SimpleFIFO Channel

```
SC_MODULE(PRODUCER) {
    sc_port< SimpleFIFOInterface<int> > master;

    SC_CTOR(PRODUCER) {
        SC_THREAD(process);
    }

    void process() {
        while(true) {
            wait(1,SC_NS);
            master->write(10);
        }
    }
};

SC_MODULE(CONSUMER) {
    sc_port< SimpleFIFOInterface<int> > slave;

    SC_CTOR(CONSUMER) {
        SC_THREAD(process);
    }

    void process() {
        while(true) {
            wait(4,SC_NS);
            cout << slave->read() << endl;
        }
    }
};
```

Create modules which have ports templated with the interface

Is This write may call an Implicit wait fn and that's why in this case we only allowed to have SC-Thread

Fraunhofer IESE

Your notes:

SimpleFIFO: A Custom Channel Example

```
int sc_main(...)
{
    PRODUCER pro1("pro1");
    CONSUMER con1("con1");
    SimpleFIFO<int> channel(4);

    pro1.master.bind(channel);
    con1.slave.bind(channel);

    sc_start(10,SC_NS);

    return 0;
}
```

Create an producer and consumer module

Create a FIFO with size 4

The Binding links the defined methods of the *Interface* with the actual implementation of the methods within in the *Channel*

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/custom_fifo

Fraunhofer IESE

Your notes:

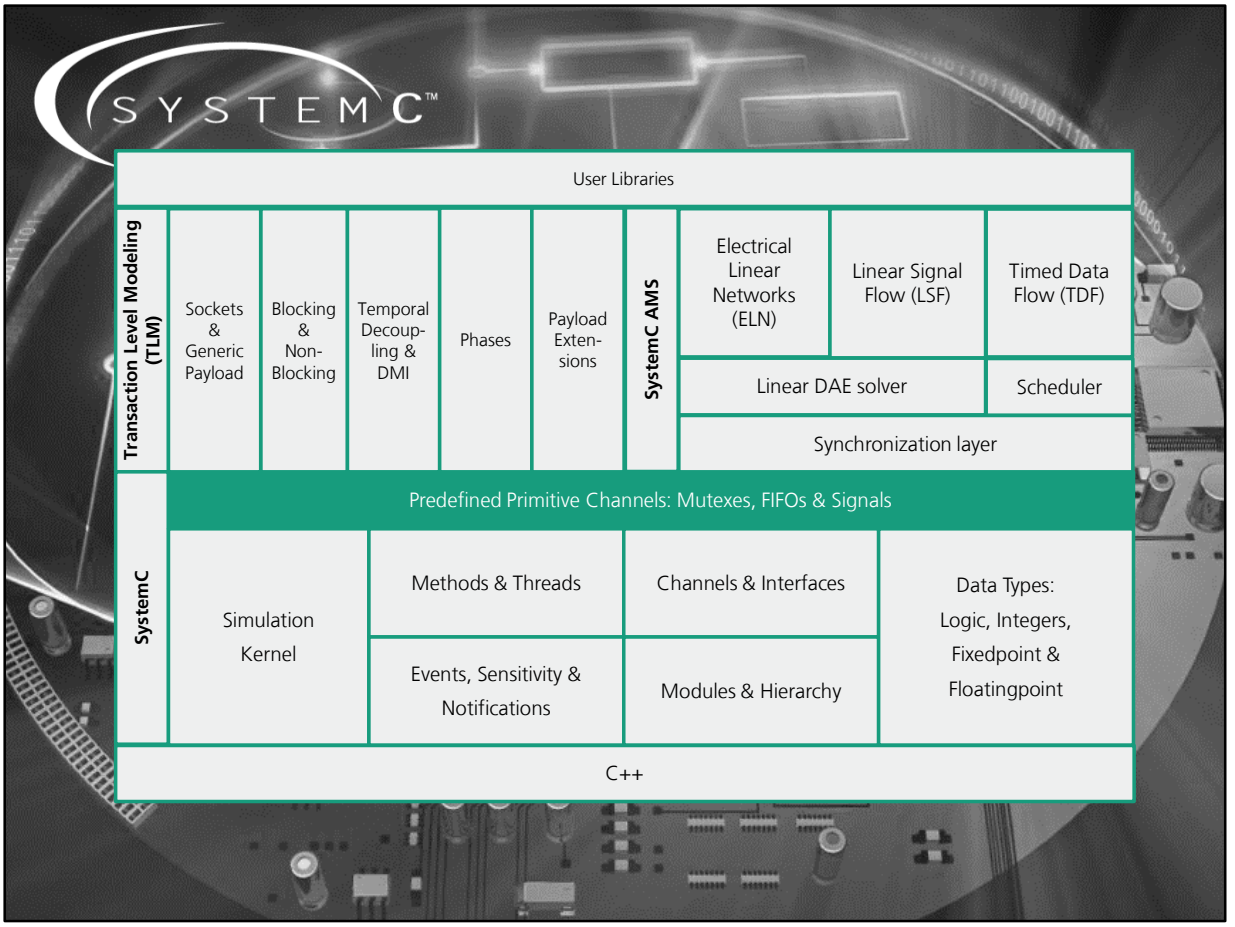
Note on Indirect Waits

This what we call indirect wait, for instance blocking read or write in SystemC FIFO invokes await when FIFO is empty or full and this case only system C thread is only allowed otherwise we will have runtime error

- Sometimes `wait()` is invoked indirectly. For instance, a blocking `read` or `write` of the `simpleFifo` (or later `sc_fifo`) invokes `wait()` when the FIFO is empty or full, respectively. In this case, the `SC_THREAD` process suspends similarly to invoking `wait` directly.
- Because `SC_METHOD` processes are prohibited from suspending internally, they may not call the `wait` method. Attempting to call `wait` either directly or implied from an `SC_METHOD` results in a runtime error. Thus, `SC_METHOD` processes must avoid using calls to blocking methods.
- For `sc_fifo`: if you want to use `sc_fifo` in a method, only use the non-blocking access methods

Fraunhofer IESE

Your notes:



Your notes:

most common SystemC

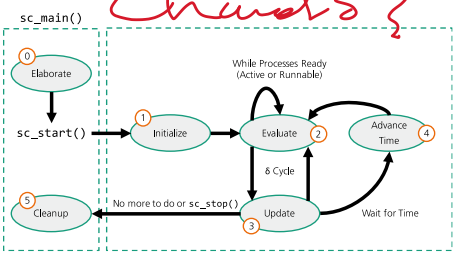
Primitive Channels

Key

- Primitive Channels allow deterministic simulation behavior:
  - Usage of Evaluate-Update-Mechanism i.e. delta cycles
  - `update_request()`, `update()`, `default_event()` (we will see later)

what are the standard system-C Primitive Channels?

- SystemC provides several Primitive Channels:
  1. `sc_signal<T>` (already known)
  2. `sc_buffer<T>`
  3. `sc_fifo<T>`
  4. `sc_mutex`
  5. `sc_semaphore`



all of them are connected to the System C Kernel.

Fraunhofer IESE

Your notes:

Yesterday we have discussed the Kernel diagram and the Primitive channels is a basic block of this, There are 3 things that have to be Implemented for Primitive channel

(1) update_request: it happens in the evaluation phase where you calculate something and then you see something changes then you call the "update".

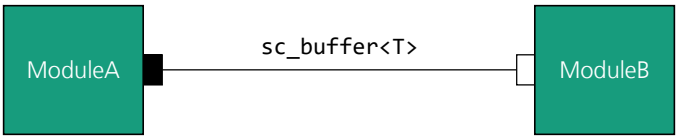
(2) Then in the update Phase "update()", the kernel will call the update function of all the channels such that they perform signal update and the sensitivity list is also handeled.

(3) Then "default_event()" have to be implemented, because when we write our signal on the sensitivity list, it will check this with the default event, The default event will return an event which is basically that is the most responsible, but there is also possibility of subevents, the rising edge and so on as you saw yesterday.

Primitive Channels: sc_buffer<T>

Buffer is derived from sc_signal, it has same methods and operators, The only difference is that in the Buffer an event is generated every time the write method is called

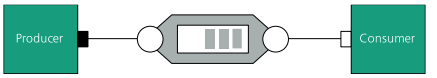
- This class is derived from **sc_signal** and has the same methods and operators
- The difference to **sc_signal** is that with **sc_buffer** an event is generated each time the **write()** method is called
- Therefore, corresponding processes sensitive to that buffer are executed.
- With **sc_signal**, an event is only generated if the old and the new value of the signal are different.



Fraunhofer IESE

Your notes:

Primitive Channels: `sc_fifo<T>`



- `sc_fifo<T>` has following predefined methods:
  - `write()`: This method writes the values passed as an argument into the FIFO. If the FIFO is full then `write()` function waits until a FIFO slot is available
  - `nb_write()`: This method is the same as `write()`, the only difference is, when the fifo is full `nb_write()` does not wait until a free FIFO slot is available. Rather it returns false.
  - `read()`: This method returns the least recent written data in the FIFO. If the FIFO is empty, then the `read()` function waits until data is available in the FIFO.
  - `nb_read()`: This method is same as `read()`, the only difference is, when the FIFO is empty, `nb_read()` does not wait until the FIFO has some data. Rather it returns false.
  - `num_available()`: This method returns the numbers of data values available in the FIFO in the current delta time.
  - `num_free()`: This method returns the number of free slots available in the FIFO in the current delta time.

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/fifo_example
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/kpn_example

Fraunhofer IESE

Your notes:

Semaphore and Mutex

we use it when we have a shared resource to avoid accessing of more than multiple to this should resource at same time.

to avoid


```
mutex.lock();
a = 1 //Shared Variable
mutex.unlock();
```


Fraunhofer IESE

Your notes:

Primitive Channels: `sc_mutex`

- With the help of a so-called Mutex (mutual exclusive), the simultaneous access of several processes to shared data structures can be regulated in software engineering.
- The primitive channel `sc_mutex` implements a corresponding lock mechanism, i.e. a mutex will be in one of two exclusive states: unlocked or locked.
- This channel is primarily intended for use with multiple processes within a module, but there is also an interface `sc_mutex_if`, so ports of this type can also be created.
- Only one process can lock a given mutex at one time. A mutex can only be unlocked by the particular process that locked the mutex, but may be locked subsequently by a different process.
- The sc_mutex class comes with pre-defined methods:
  - `int lock()`: Lock the mutex if it is free, else wait till mutex gets free
  - `int unlock()`: Unlock the mutex, returns -1 if mutex was not locked
  - `int trylock()`: Check if mutex is free, if free then lock it else return -1.

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/mutex_example

Fraunhofer IESE

Note: The request-update mechanism is not used for the mutex ports!

Your notes:

Primitive Channels: sc_semaphore

- A semaphore is an extension of the simple mutex.
- An additional integer value is introduced (called semaphore value), which is set to the permitted number of concurrent accesses when the semaphore is constructed. A semaphore with a value of 1 is therefore a mutex.
- The semaphore class `sc_semaphore` also has an interface.
- sc_semaphore has following predefined methods:
  - `int wait()`: If the semaphore value is equal to 0, the member function `wait` shall suspend until the semaphore value is incremented (by another process), at which point it shall resume and attempt to decrement the semaphore
  - `int trywait()`: If the semaphore value is equal to 0, the member function `trywait` shall immediately return the value -1 without modifying the semaphore value
  - `int post()`: increments the semaphore value. If processes exist that are suspended and are waiting for the semaphore value to be incremented, exactly one of these processes shall be permitted to decrement the semaphore value (the choice of process being non-deterministic) while the remaining processes shall suspend again
  - `int get_value()`: returns value the semaphore

Fraunhofer IESE

Note: The request-update mechanism is not used for the semaphore ports!

Your notes:

Why Virtual Base Class Concept for Channels?

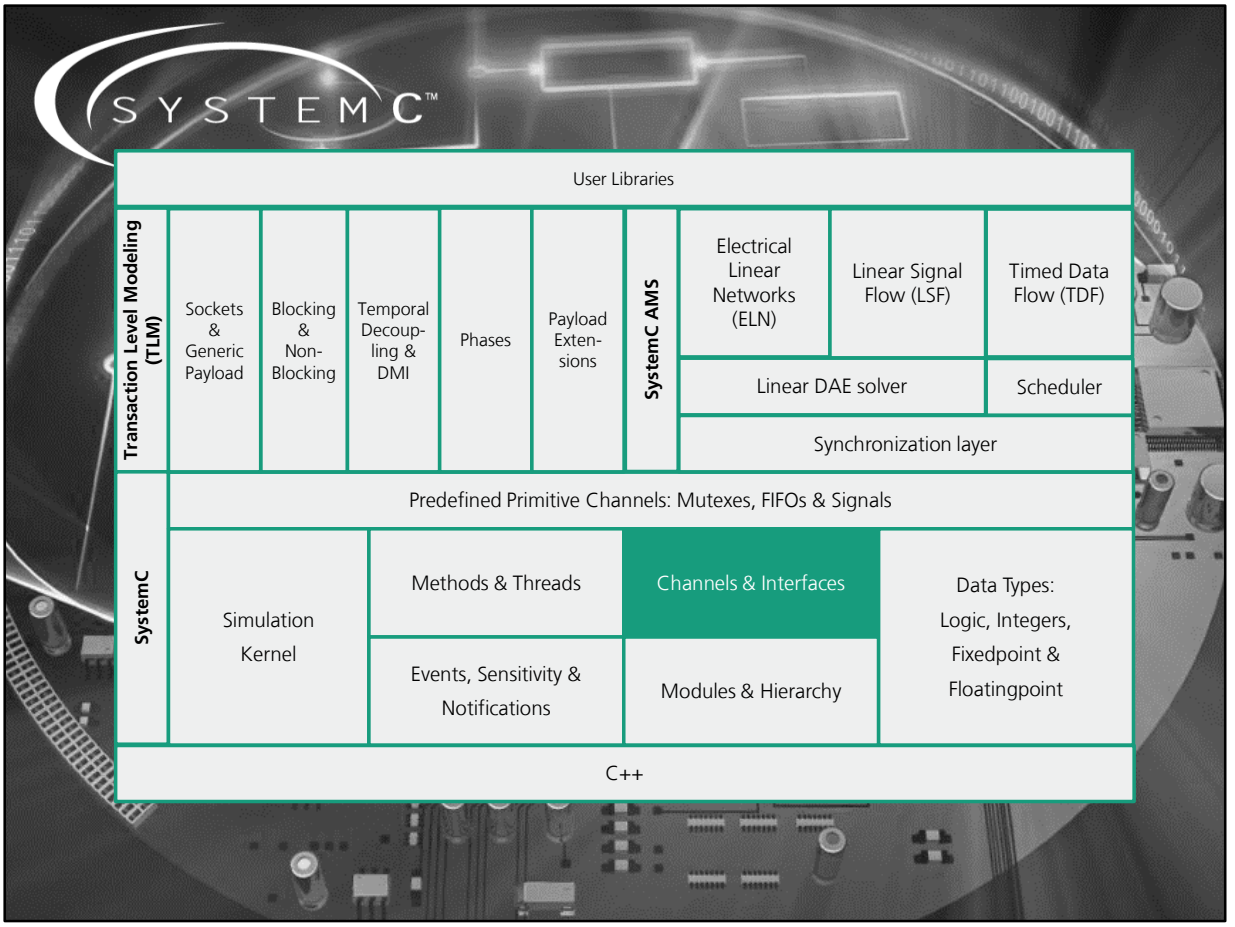
- To provide variability and interoperability in modeling
- Example 2 memory channels with same interface but different implementation:

```
class memorySimple: public memoryInterface {
    public:
    void write(unsigned int addr, int data)
    {
        mem[addr] = data;
    }
    void int read(unsigned int addr)
    {
        return mem[addr];
    }
    private:
    int mem[1024];
}
```

```
class memoryDetail: public memoryInterface {
    public:
    void write(unsigned int addr, int data)
    {
        // Complex implementation of write
    }
    void int read(unsigned int addr)
    {
        // Complex implementation of write
    }
    private:
    // Complex implementation …
}
```


Fraunhofer IESE

Your notes: If we change the bus from CAN to SPI, we don't have to change Producer and consumer modules to fit this we just change the channel Type and keeping our modules same

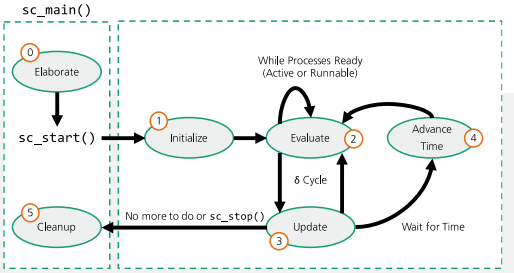


Your notes:

customized systemC channels

How Primitive channels works :-

Signal: A Custom Primitive Channel with Evaluate Update Mechanism




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

using namespace std;

template <class T>
class SignalInterface : public sc_interface
{
    public:
    virtual T read() = 0;
    virtual void write(T) = 0;
};
```

33

Fraunhofer IESE

Your notes:

what i want to build now with you is our own Custom signal which do the same thing as system-C signal but it's important to me to understand how the kernel works -

① we create an interface and then we derive our signal from that interface

② we want to build the signal as it's in systemc, Therefore it is a primitive channel, and PrimitiveChannel allows us to put things in the sensitivity list of, sowe want later on, so we can put our signal on a sensitivity list of a process of a module then in order for this to happen we have to derive from another class "sc_PrimeChannel" which stands for primitive channels,

③ we see in that class some Pure virtual functions such as update & default event in the kernel.

④ inside the signal we define the current value of signal and the new value of signal and data

we create an event "value changes event" this event will be added to sensitivity list, so every time the value changes we will Fire this event then the systemc kernel will notice there is something here happened to the signal.

(5) Then we define the constructor, and we implements also the read and write fn

(6) in the write fn we check if the new value is equal the current value then that means there is a change then we inform systemc kernel we have an update Please take it into consideration.

(7) Then we implement the update value fn

Signal concept + Implementation

Signal: A Custom Primitive Channel with Evaluate Update Mechanism

```
template <class T>
class Signal : public SignalInterface<T>,
               public sc_prim_channel
{
    private:
    T currentValue;
    T newValue;
    sc_event valueChangedEvent;

    public:
    Signal() {
        currentValue = 0;
        newValue = 0;
    }

    T read()
    {
        return currentValue;
    }

    void write(T d)
    {
        newValue = d;
        if(newValue != currentValue)
        {
            // Call to SystemC Scheudler
            request_update();
        }
    }
```

```
    void update() // MUST be implemented!
    {
        if(newValue != currentValue)
        {
            currentValue = newValue;
            valueChangedEvent.notify(SC_ZERO_TIME);
        }
    }

    const sc_event& default_event() const // Should be!
    {
        return valueChangedEvent;
    }
};
```

here we apply changes

- Declare interface as usual
- Derive from `sc_prim_channel`
- Implement `update()` function
- Implement `default_event()` function
- Later: *Event Finders*

we are trying to implement SystemC signal with implicit update

here we detect change

↳ system kernel get notified that the value changes

Fraunhofer IESE

Your notes:

Signal: A Custom Primitive Channel with Evaluate Update Mechanism

```
SC_MODULE(PRODUCER) {
    sc_port< SignalInterface<int> > master;

    SC_CTOR(PRODUCER) {
        SC_THREAD(process);
    }

    void process() {
        master->write(10);
        wait(10,SC_NS);
        master->write(20);
        wait(20,SC_NS);
        sc_stop();
    }
};

SC_MODULE(CONSUMER) {
    sc_port< SignalInterface<int> > slave;

    SC_CTOR(CONSUMER) {
        SC_METHOD(process);
        sensitive << slave;
        dont_initialize();
    }

    void process() {
        int v = slave->read();
        std::cout << v << std::endl;
    }
};
```

Sensitive to default_event() !

```
int sc_main(…)
{
    PRODUCER pro1("pro1");
    CONSUMER con1("con1");
    Signal<int> channel;

    pro1.master.bind(channel);
    con1.slave.bind(channel);

    sc_start(sc_time(100,SC_NS));

    return 0;
}
```

Try code on github:

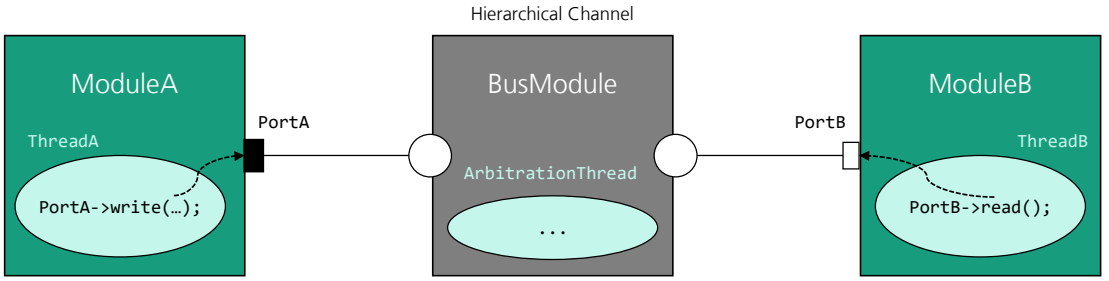
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/custom_signal

Fraunhofer IESE

Your notes:

Hierarchical Channels

- Primitive channels are derived from `sc_prim_channel` and are "*pasive*"
- Hierarchical Channels are derived from `sc_module` and can be "*active*"
  - Hierarchical Channels use also the concept of Interfaces
  - They can have internal `SC_THREAD`s and `SC_METHOD`s
  - They can consist of other `sc_module`s, fw ports to outside `sc_export`
- Heavily used in TLM
- Hierarchical Channels do not have the "Evaluate-Update Mechanism"



Fraunhofer IESE

Your notes:

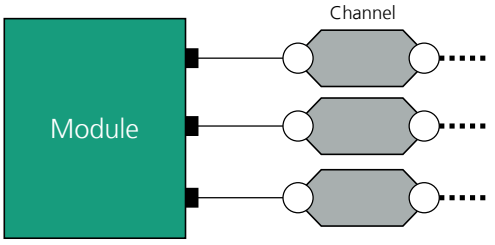
If you want to model a CAN Bus where you have some logic also within the bus, something that does Arbitration or does a decision on routing or whatever, so that make sense to have somehow logic in your module that's what we called Heirarichal channel.

Heirarichal channel is notonly derived From the Prime Channel but it may be also derived From SC-Module, so it's kinda a mix of Channel and module, then It can be active and Heirarchal Channels can use the concept of interface but they can also to implement intenally SC-Threads & SC-Methods and they can consists of other systemc modules, you can put a CPU inside your Bus Module for ex if you want this, there is no limitation here.

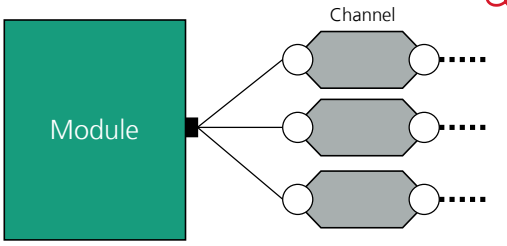
→ The Heirarchal Channels concept is the main idea of the Transaction level Modeling, it's heavily used in TLM where we will communicate by Passing classes

and Packets, we have then the Hirachl Channel, They don't have an update mechanism.

Ports, Port Arrays and Multiports



- Static declaration of ports and binding to separated channels.
- Is fixed on compile time
- Port arrays are more convenient

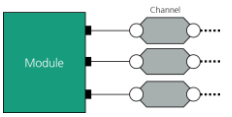


- Dynamic port creation during elaboration phase
- Using Multiports

Fraunhofer IESE

Your notes:

Port Arrays




```
SC_MODULE(module) {
    // Instead of
    //sc_port<sc_fifo_out_if<int> > port1;
    //sc_port<sc_fifo_out_if<int> > port2;
    //sc_port<sc_fifo_out_if<int> > port3;

    sc_port<sc_fifo_out_if<int> > port[3];

    SC_CTOR(module){
        SC_THREAD(process);
    }

    void process() {
        for(int i=0; i < 3; i++) {
            port[i]->write(2);
            std::cout << "Write to port " << i
                      << std::endl;
            wait(1, SC_NS);
        }
    }
};
```

```
int sc_main(…)
{
    module m("m");
    sc_fifo<int> f1, f2, f3;

    m.port[0].bind(f1);
    m.port[1].bind(f2);
    m.port[2].bind(f3);

    sc_start();
    return 0;
}
```

- Static port creation at compile time
- Connected channels are addressed with [ ] operator

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/portarrays

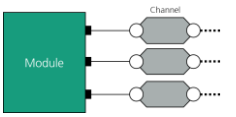
Fraunhofer IESE

Your notes:

here is Templated Port Arrays.

Templated Port Arrays

how can we bind this.?



By default i have 1 Port but you can change it later

```
template <int N=1>
SC_MODULE(module)
{
    sc_port<sc_fifo_out_if<int> > port[N];

    SC_CTOR(module){
        SC_THREAD(process);
    }

    void process() {
        for(int i=0; i < N; i++)
        {
           port[i]->write(2);
           std::cout << "Write to port "
                     << i << std::endl;
           wait(1, SC_NS);
        }
    }
};
```

my module should have 3 Ports

```
int sc_main(…)
{
    module<3> m("m");
    sc_fifo<int> f1, f2, f3;

    m.port[0].bind(f1);
    m.port[1].bind(f2);
    m.port[2].bind(f3);

    sc_start();
    return 0;
}
```

we instantiate our module

Then i may use 3 of the system FIFO

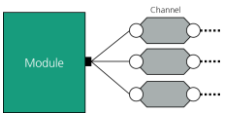
But This Static where we define our module have this no. of Ports

- Static port creation at compile time
- Using Template Parameter
- Connected channels are addressed with [ ] operator

Fraunhofer IESE

Your notes:

Multiports




```
SC_MODULE(module){
    sc_port<sc_fifo_out_if<int>,
    0,
    SC_ZERO_OR_MORE_BOUND> port;

    SC_CTOR(module){
        SC_THREAD(process);
    }

    void process(){
        for(int i; i < port.size(); i++){
            port[i]->write(2);
            std::cout << "Write to port "
                      << i << std::endl;
        }
    }
};
```

```
int sc_main(…)
{
    module m("m");
    sc_fifo<int> f1, f2, f3;

    m.port.bind(f1);
    m.port.bind(f2);
    m.port.bind(f3);

    sc_start();
    return 0;
}
```

May lead to out of range error during runtime!

- Dynamic port creation during elaboration phase using Multiports
- Connected channels are addressed with `[]` operator
- Number of bound channels with `size()` method

Try code on github:

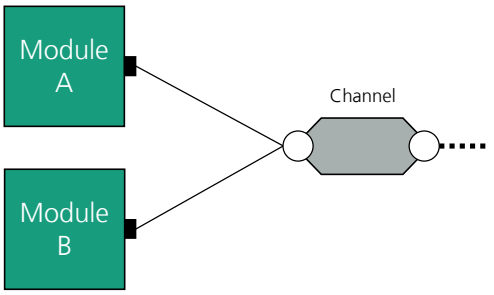
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/multiports

Fraunhofer IESE

Your notes:

If you want to do some Analysis to see when your system have one Port or increase more Ports would be better so we use it in analysis.

Multiple Bindings




```
int sc_main(…)
{
    module a("a");
    module b("b");

    myChannel<int> c;

    a.port.bind(c);
    b.port.bind(c);

    sc_start();
    return 0;
}
```

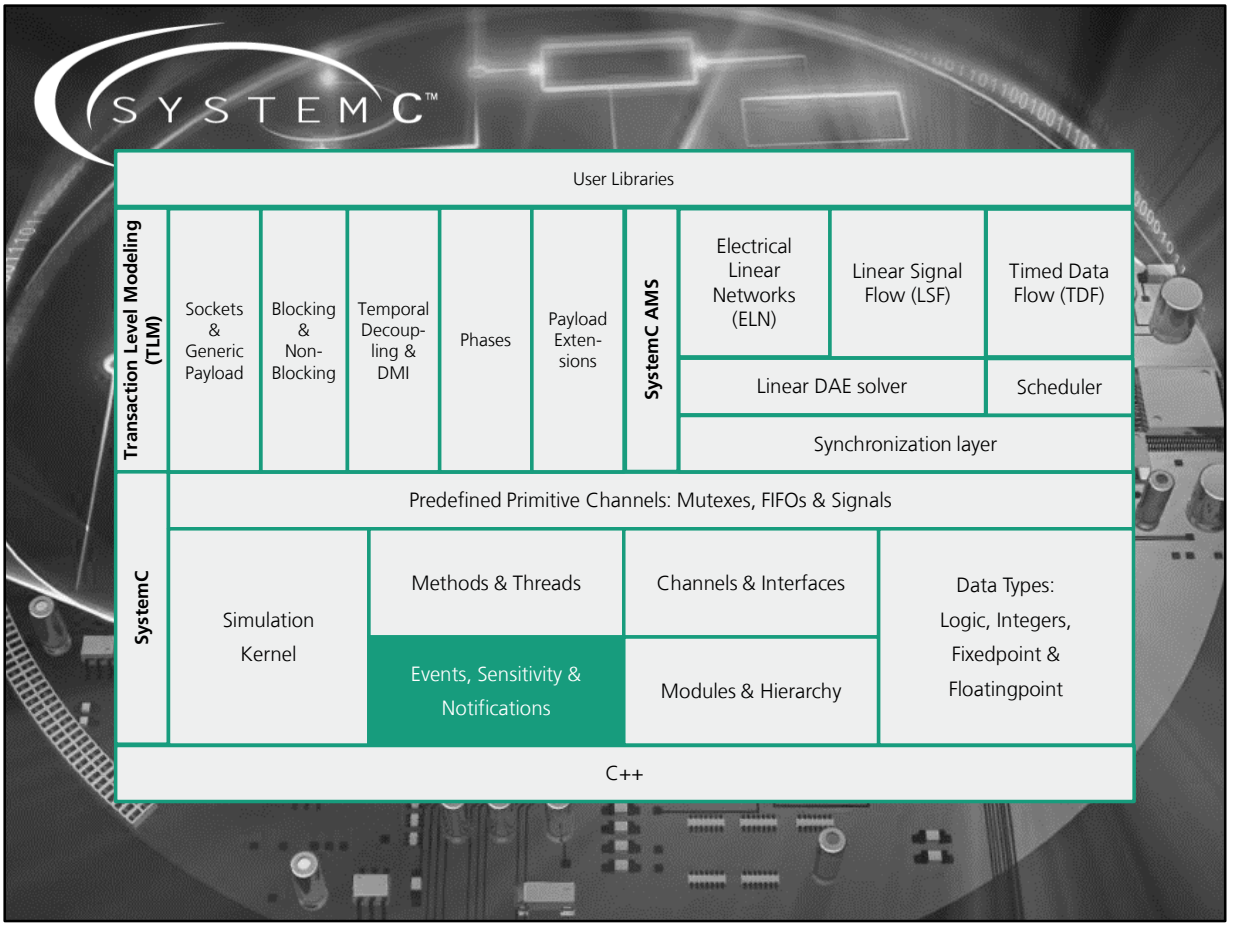
- Works in general
- `sc_fifo`, `sc_signal` … can have only one writer

Fraunhofer IESE

Your notes:

This also works, there are some limitation, but lets see our code here, if we have a custom channel. You can also bind several Producer to them, that works in general, but there is a But in RTL modeling style fo sc_fifo, sc_signal only allowed to have only one module that writes to it, so you that only one driver is allowed for signal in RTL, But it's allowed in General in systemC - we get away from RTL So we can Find some use cases where we define a specific Channel then we connect

No eA & B to the channel we will do that in TLM.



Your notes:

SystemC Events: sc_event

- Events are implemented with the `sc_event` class.
  - `sc_event myEvent;`
- Events are caused or fired through the event class member function `notify()`:
  - `myEvent.notify();`
    Avoid: events can be missed, non-determinism!
    Event is notified in the current evaluation phase
  - `myEvent.notify(SC_ZERO_TIME);`
  - `myEvent.notify(time);`
  - `myEvent.notify(10,SC_NS);`
  - `myEvent.cancel();`
- Only the first notification is noted

```
void triggerProcess() {
  wait(SC_ZERO_TIME);
  triggerEvent.notify(10,SC_NS);
  triggerEvent.notify(20,SC_NS); // Will be ignored
  triggerEvent.notify(30,SC_NS); // Will be ignored
}
```

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/sc_event_and_queue

Fraunhofer IESE

Your notes:

SystemC Events: sc_event_queue

```
SC_MODULE(eventQueueTester) {
    sc_event_queue triggerEventQueue;

    SC_CTOR(eventQueueTester) {
        SC_THREAD(triggerProcess);
        SC_METHOD(sensitiveProcess);
        sensitive << triggerEventQueue;
        dont_initialize();
    }

    void triggerProcess() {
       wait(100,SC_NS);
       triggerEventQueue.notify(10,SC_NS);
       triggerEventQueue.notify(20,SC_NS);
       triggerEventQueue.notify(40,SC_NS);
       triggerEventQueue.notify(30,SC_NS);
    }
    void sensitiveProcess() {
        cout << "@" << sc_time_stamp() << endl;
    }
};
```

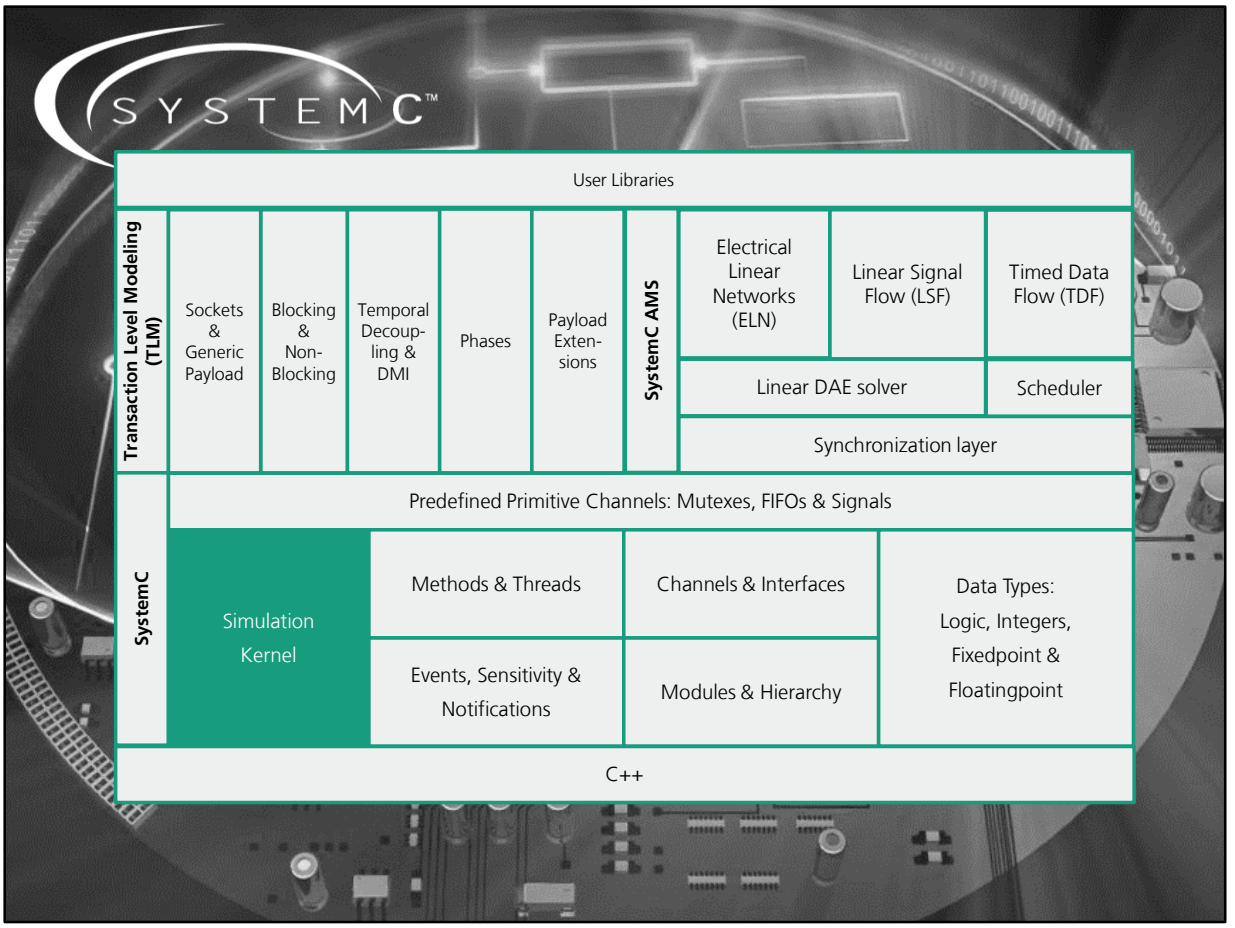
- The class `sc_event_queue` notes all notifications
- Orders events w.r.t ascending time
- Provides also interface `sc_event_queue_if` for using as a port

Output:
@110ns
@120ns
@130ns
@140ns

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/sc_event_and_queue

Fraunhofer IESE

Your notes:



Your notes:

We have seen already Methods & Threads, They have been created at the elaboration phase during compile

Time But System C Provides Something called Dynamic Process where we can create new process during Runtime Simulations

Dynamic Processes

- So far processes (threads and methods) were created during elaboration
- SystemC allows to generate new *dynamic* processes during simulation
- Fields of applications: Of creating Dynamic processes during Runtime.
  - Testbenches
  - Verification scenarios.
  - Modeling of SW
  - Modeling of OS You can model RTOS by using System C process because you can model concurrency and concurrent task and with the wait statement you can model Preemption and Task Priorities That make sense that you can
- Enabled by using `#define SC_INCLUDE_DYNAMIC_PROCESSES` before `#include <systemc.h>`, or using a compiler flag
- Creation of process by function `sc_spawn()`
- Allows passing of arguments for processes!

If you want to use this you have to enable these in your code or in your C-make file you have to enable it and make file

Fraunhofer IESE

Your notes:

- You can create new process called SC_SPAWN().

Also used in modeling of RTOS.

during Simulation we can create new process in your ... sometimes it's useful ... verification or ...

Dynamic Processes

We don't need to go into the code details understand the ideas only,

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <iostream>
#include <systemc.h>
using namespace std;

SC_MODULE(module) {
    SC_CTOR(module){
        SC_THREAD(parentProcess);
    }
    void parentProcess() {
        wait(10,SC_NS);
        sc_process_handle handle = sc_spawn(
            sc_bind(&module::childProcess, this, 5)
        );
        wait(handle.terminated_event());
    }
    void childProcess(int id) {
        cout << id << " started" << endl;
        wait(10,SC_NS);
    }
};
```

parentProcess can create new Processes

Dynamic

Functional Pointer to child process is pur under you Pass

```
int sc_main(…)
{
    module m("m");
    sc_start();
    return 0;
}
```

- Handle process with sc_process_handle
- sc_spawn uses sc_bind in order to reference to dynamic method
- Dynamic processes have an termination event
- Arguments

we can use

Try code on github: https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/dynamic_processes

↳ This Process can also use wait Process

Fraunhofer IESE

= we have Systemc Module then Parent Process and the Parent Process can create new processes.

Your notes:

- we have a function Pointer to the child process which is also like the Parent Process that is defined down here,

Report Handling

- SystemC provides a centralized way for reporting on the terminal
  - `SC_REPORT_INFO("id", "Message")`: print some information
  - `SC_REPORT_WARNING("id","Meassage")`: Warning, which to a possible problem
  - `SC_REPORT_ERROR("id","Meassage")`: Serious Problem, exeption is thrown which can be handled by try{}catch{} and the simulation continues
  - `SC_REPORT_FATAL("id","Meassage")`: Serious unsolvable problem, the simulation is stopped
- `sc_assert()`
  - If argument is false, then simulation is stopped like for `SC_REPORT_FATAL`

Fraunhofer IESE

Your notes:

Report Handling Example

```
SC_MODULE(module) {
  bool c1;
  bool c2;

  SC_CTOR(module) {
    c1 = true;
    c2 = true;

    sc_assert(c1 == true && c2 == true);

    SC_REPORT_INFO("main","Report ...");
    SC_REPORT_WARNING("main","Report ...");
    try {
       SC_REPORT_ERROR("main","Report ...");
    }
    catch(sc_exception e){
       cout << "what:" << e.what() << endl;
    }
    SC_REPORT_FATAL("main","Report & Stop...");
  }
};
```

```
int sc_main(…)
{
  // Optional: Console otherwise ...
  sc_report_handler::set_log_file_name("out.log");
  sc_report_handler::set_actions(SC_INFO, SC_LOG);
  sc_report_handler::set_actions(SC_WARNING, SC_LOG);

  module m("m");

  sc_start();
  return 0;
}
```

```
        SystemC 2.3.1-Accellera --- Feb 25 2016 17:15:15
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED

Info: main: Report Info...

Warning: main: Report Warning...
In file: main.cpp:18
do some handling for std::exception

Fatal: main: Report Error and Stop...
In file: main.cpp:25
```

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/reporting

Fraunhofer IESE

Your notes: It's helpful to do it

You can customize your report handler

Custom Reporthandler

```
void reportHandler(const sc_report &report,
                   const sc_actions &actions)
{
  [...]
  switch(report.get_severity()) {
    case SC_INFO    : severity = "INFO   "; break;
    case SC_WARNING : severity = "WARNING"; break;
    case SC_ERROR   : severity = "ERROR  "; break;
    case SC_FATAL   : severity = "FATAL  "; break;
  }
  std::ostream& stream = std::cout;
  stream << report.get_time()
         << " + "  << sc_delta_count() << "Î´"
         << " "    << report.get_msg_type()
         << " : [" << severity << "] "
         << ' '    << report.get_msg()
         << " (File: "<< report.get_file_name()
         <<" Line: "
         << report.get_line_number() << ")"
         << std::endl;
  [...]
}
```

```
int sc_main(…)
{
    sc_core::sc_report_handler
           ::set_handler(reportHandler);
    module m("m");

    sc_start();
    return 0;
}
```

time future error happening Info.

```
        SystemC 2.3.1-Accellera --- Feb 25 2016 17:15:15
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
0 s + 0δ main : [INFO    ]  Report Info... (File: main.cpp Line: 17)
0 s + 0δ main : [WARNING ]  Report Warning... (File: main.cpp Line: 18)
0 s + 0δ main : [ERROR   ]  Report Error... (File: main.cpp Line: 20)
0 s + 0δ main : [FATAL   ]  Report Error and Stop... (File: main.cpp Line: 25)
Abort trap: 6
```

- Use custom reporthandler
- For more application/simulation specific output

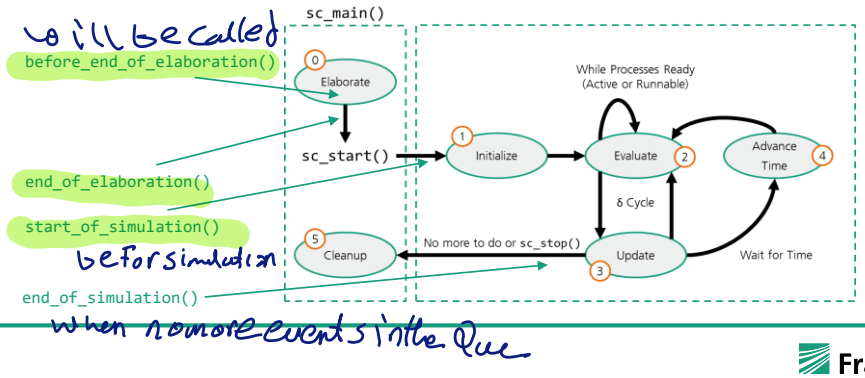
Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/reporting

Fraunhofer IESE

Your notes:

Callbacks

- The classes `sc_module`, `sc_prim_channel`, `sc_port` and `sc_export` define 4 virtual callback functions:
  - `before_end_of_elaboration()`
  - `end_of_elaboration()`
  - `start_of_simulation()`
  - `end_of_simulation()`
- If a module implements one of these functions, the scheduler will call them!
- Separation of debugging and functionality



Fraunhofer IESE

Your notes:

So in the classes, system C module, Prime Chanel, Port - an deXPort, There are 4 virtual call back Functions

- Because we dont have stuff in our Constructors and only can keep the logic Part init, and any thing that is there for debugging and Printing out any observations can be done

Callbacks

- `before_end_of_elaboration()`

  In this callback function, it is possible to instantiate further SystemC objects such as modules, channels or ports or to make port bindings and thus subsequently change the module hierarchy. Furthermore, other processes can be registered for the scheduler, which are static.

- `end_of_elaboration()`

  This callback function is called after all callbacks of **`before_end_of_elaboration()`** have been executed. This ensures that all bindings are present and the module hierarchy is complete. Therefore, it is no longer allowed to add other SystemC objects, such as modules, channels or ports, or to make bindings. However, dynamic processes can be logged on to the scheduler here. Furthermore, diagnostic messages can be printed.

Fraunhofer IESE

Your notes:

Callbacks

- `start_of_simulation()`

  This function is executed after calling `sc_start()`, text or trace files can be opened or diagnostic messages can be printed. Furthermore, it is still possible to register dynamic processes at the scheduler.

- `end_of_simulation()`

  This function is only executed when the simulation is terminated by calling `sc_stop()` by the user. If the simulation is terminated without calling the `sc_stop()` function (no pending events for the scheduler) then this function is not called. In this function, for example, text or trace files can be closed again.

  The desctuctors are called after this call.

Fraunhofer IESE

Your notes:

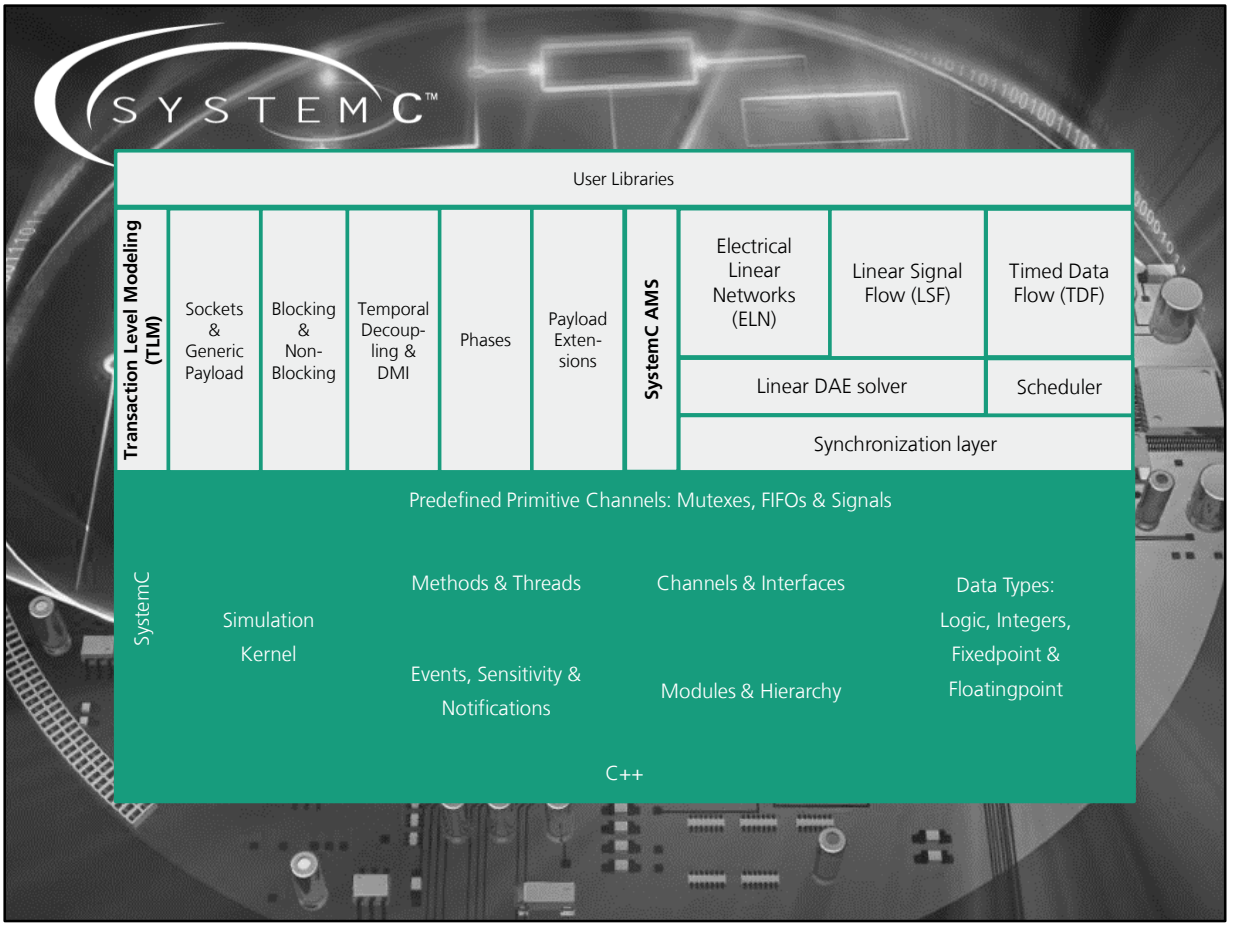
Callbacks

```
SC_MODULE(module){
    public:
    sc_in<bool> clk;
    sc_trace_file *tf;

    SC_CTOR(module){}

    void process(){
        wait(5);
        sc_stop();
    }

    void before_end_of_elaboration() {
        cout << "before_end_of_elaboration"
             << endl;
        SC_THREAD(process);
        sensitive << clk.pos();
    }

    void end_of_elaboration() {
        cout << "end_of_elaboration" << endl;
    }

    void start_of_simulation() {
        cout << "start_of_simulation" << endl;
        tf = sc_create_vcd_trace_file("trace");
    }

    void end_of_simulation() {
        cout << "end_of_simulation" << endl;
        sc_close_vcd_trace_file(tf);
    }
};
```

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/callbacks

Fraunhofer IESE

Your notes:



Your notes: