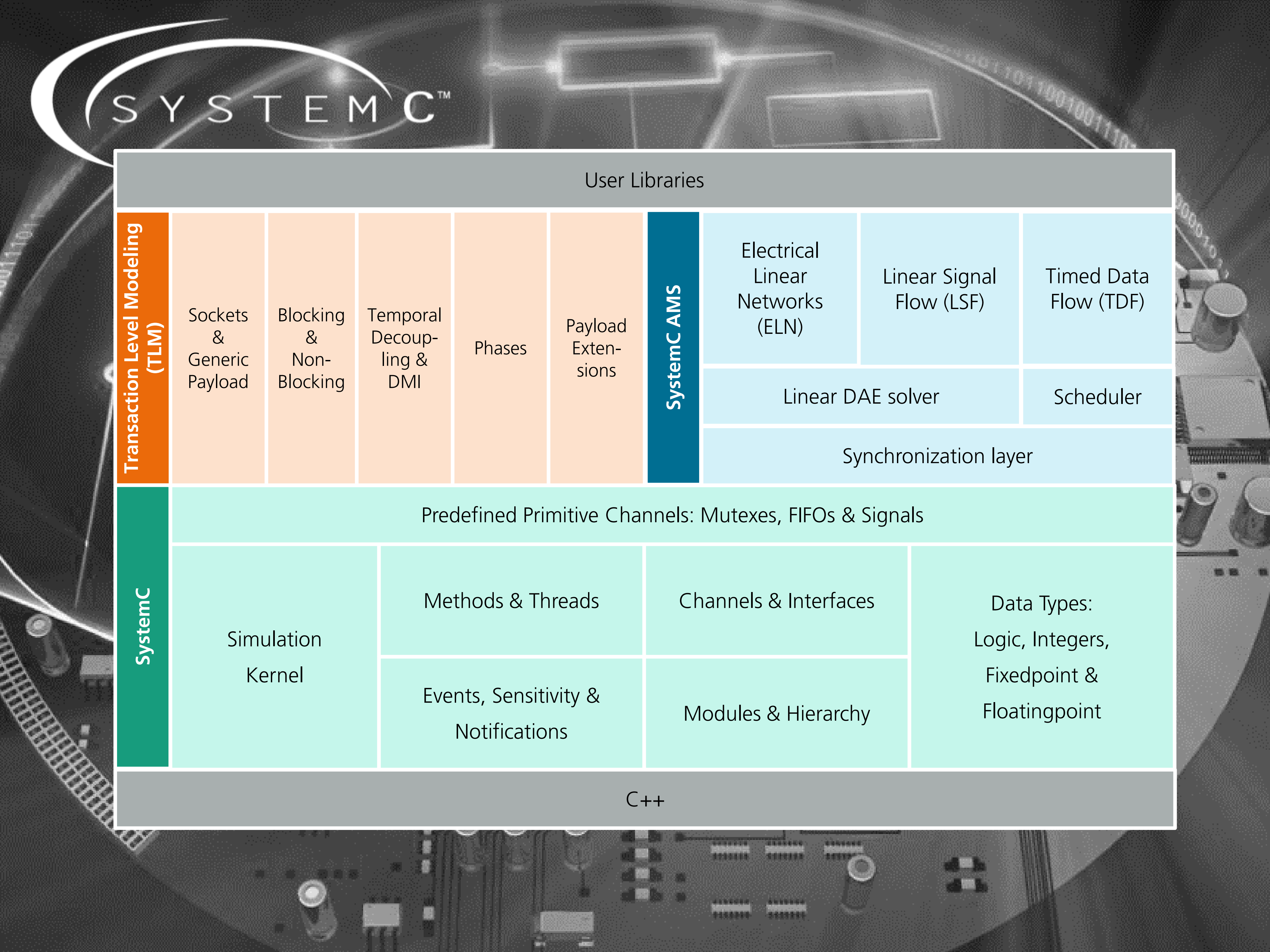
SystemC and Virtual Prototyping

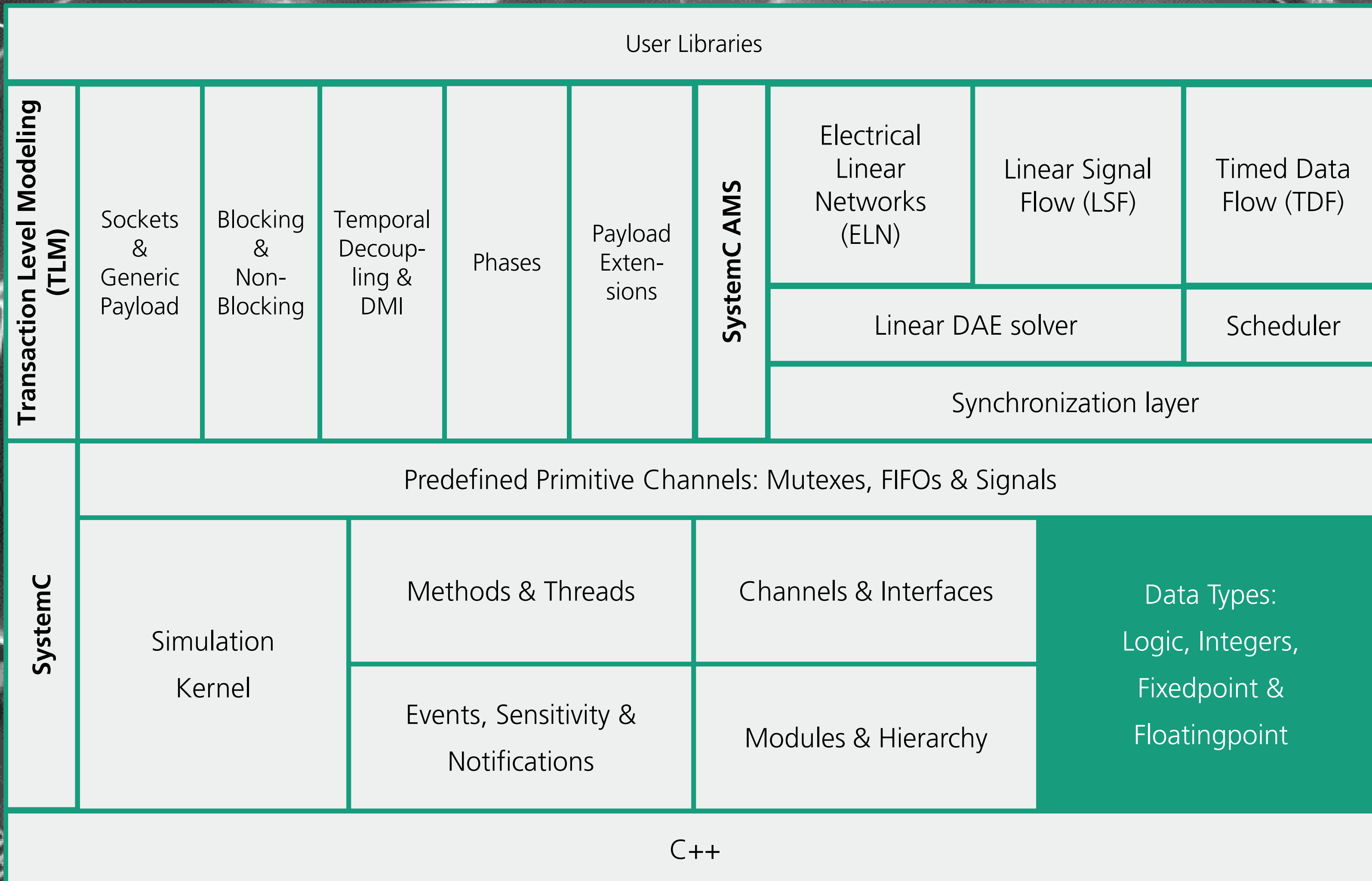
Dr. Matthias Jung, Fraunhofer Institute IESE

matthias.jung@iese.fraunhofer.de

Fraunhofer
IESE

SystemC Advanced


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++


User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Events, Sensitivity & Notifications
Modules & Hierarchy
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
C++

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

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

SystemC Logic Datatypes: `sc_bit`, `sc_bv<W>`

```
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
```

- `sc_bit`:
  deprecated, use bool instead!

- `sc_bv<W>`: bit vector
  - Width as template parameter
  - Typical operators overloaded: `&,|,^,~, …`
  - `X_reduce()` methods
  - Ranges
  - Concatenation
  - Similar VHDL's `bit_vector`

Try code on github:

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

Fraunhofer IESE

SystemC Logic Datatypes: sc_logic, sc_lv<W>

- sc_logic features 4 States:
  - 'X': Unknown
  - '0': 0
  - '1': 1
  - '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.

Remember: Fixed Point and Two's Complement Numbers

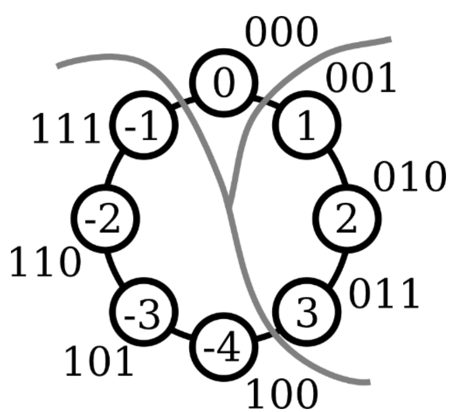
- Positive Fixed point:

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

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

- Two's Complement:

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



- No double 0
- Asymmetric range
- Simple hardware performing (add, sub …)

Fraunhofer IESE

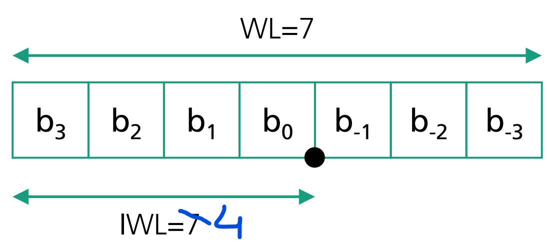
SystemC Integer Datatypes: `sc_int<W>, sc_uint<W>, …`

- `sc_int<W>` for signed integers and `sc_uint<W>` for unsigned integers
  - Provides efficient way to model data with specific widths (1-64)
  - 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.

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

SystemC Fixpoint Datatypes: `sc_fixed<…>`, …

- `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>:`

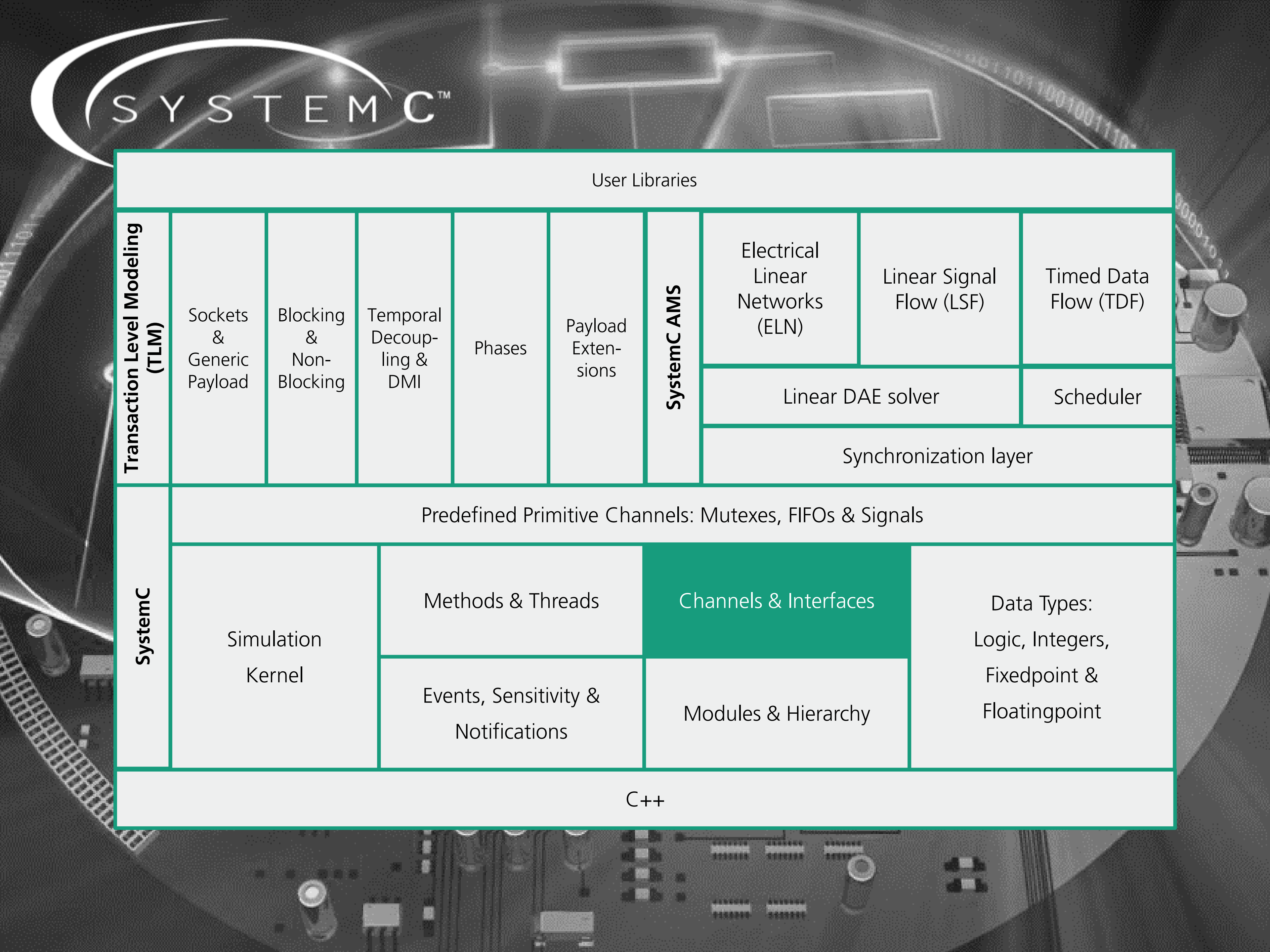


WL = Word Length

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


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++

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;
}
```

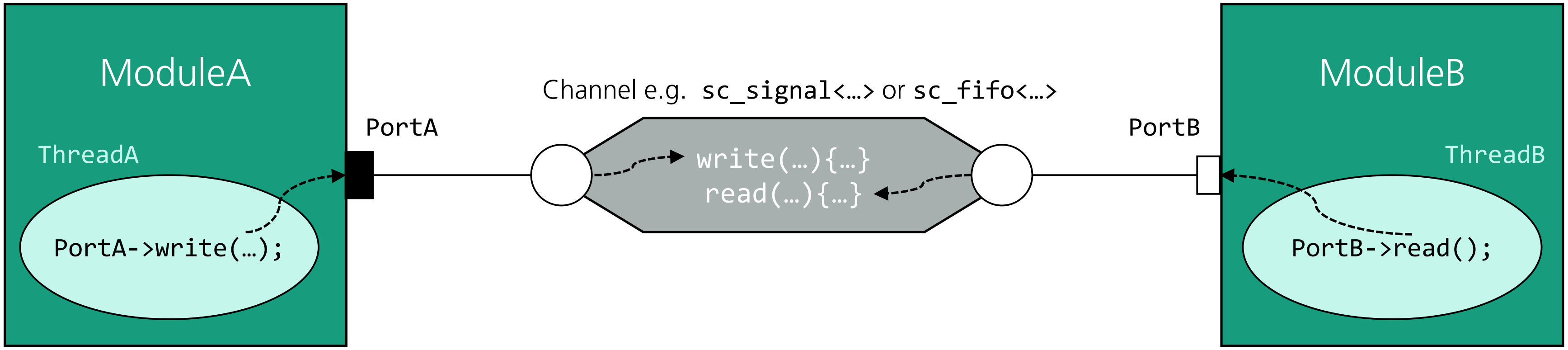
Output:
Rectangle class area: 50
Triangle class area: 25

- 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

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*



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

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

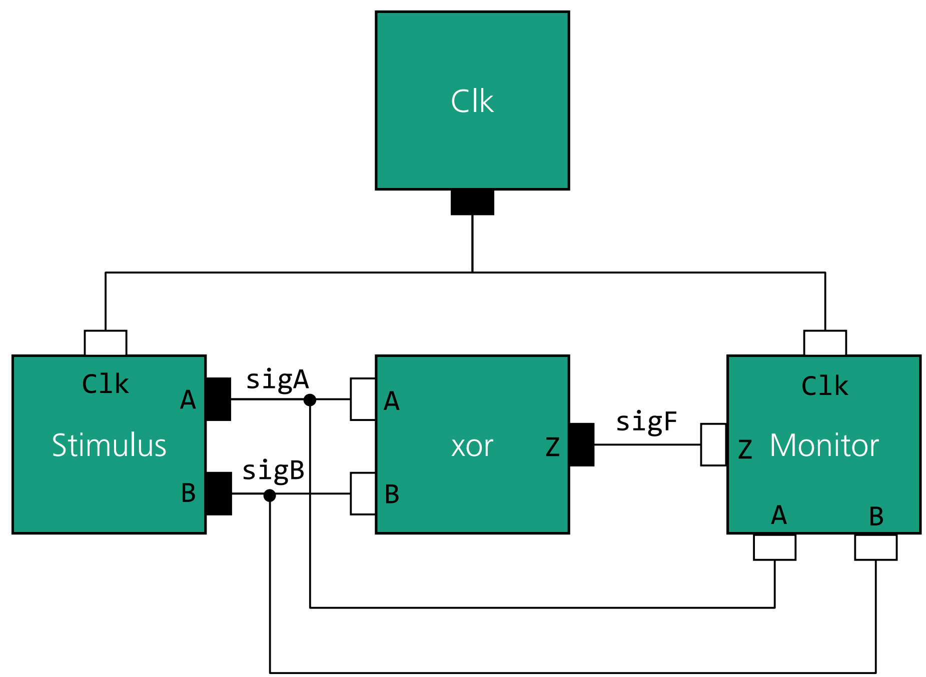
- Specialized ports `sc_in`, `sc_out`, `sc_inout` for `sc_signal`, for RTL modelling and easy use.
- sc_port has several parameters:
  - Interface (required)
  - N (optional): max number of channels to be bound
  - Policy (optional):
    - `SC_ONE_OR_MORE_BOUND`
    - `SC_ZERO_OR_MORE_BOUND`
    - `SC_ALL_BOUND`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

Fraunhofer IESE

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;
}
```

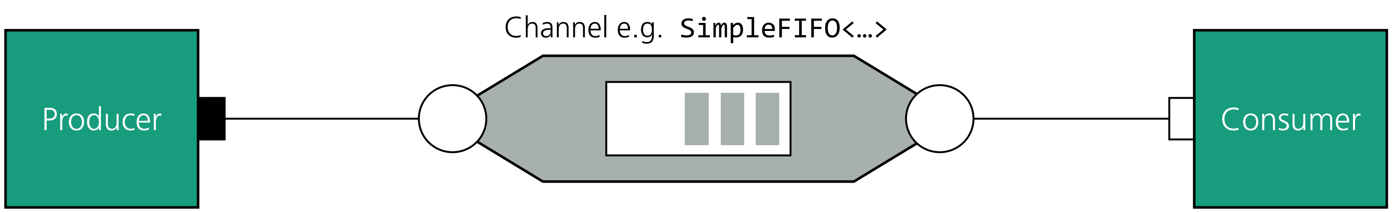
- The methods are implemented in the channel
- Binding the channel to the port during runtime: A port forwards the calls of the interface methods in the module to the channel that was bound to the port.



Fraunhofer IESE

SimpleFIFO: A Custom Channel Example

- 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

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

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

Fraunhofer IESE

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

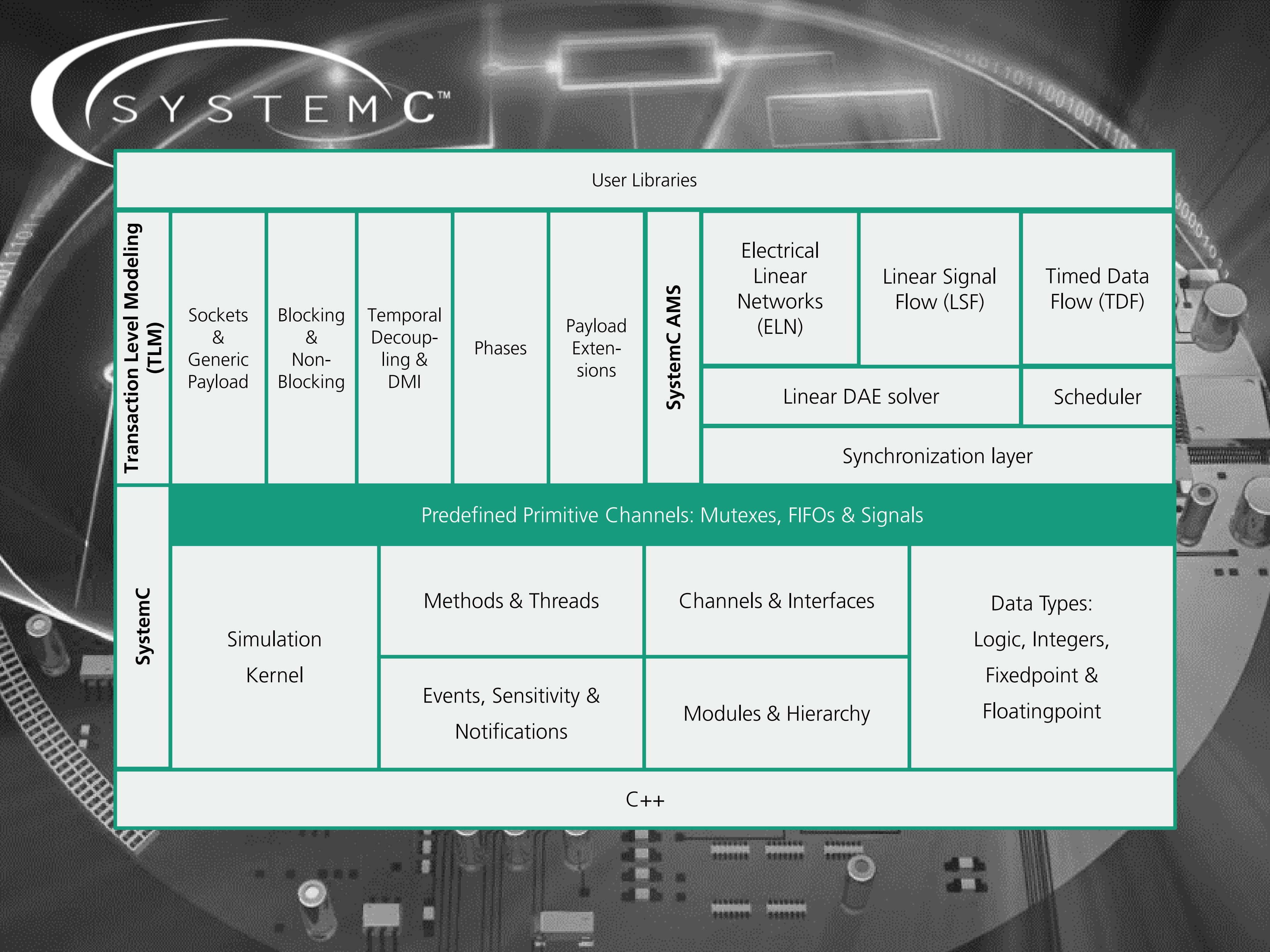
Note on Indirect Waits

- 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

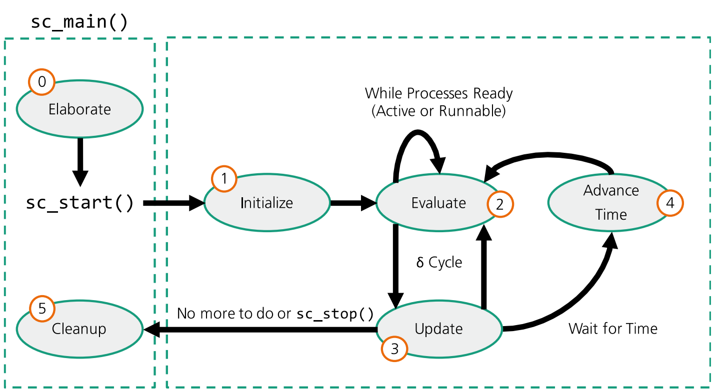
Fraunhofer IESE


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++

Primitive Channels

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

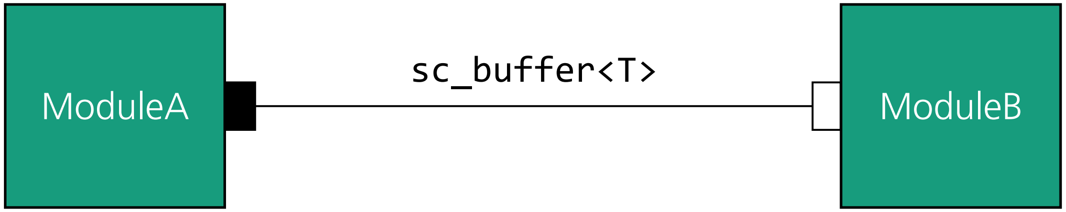
- SystemC provides several Primitive Channels:
  - `sc_signal<T>` (already known)
  - `sc_buffer<T>`
  - `sc_fifo<T>`
  - `sc_mutex`
  - `sc_semaphore`



Fraunhofer IESE

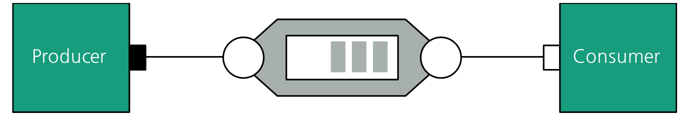
Primitive Channels: `sc_buffer<T>`

- 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

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 <u>waits</u> 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 <u>waits</u> 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

Semaphore and Mutex


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


Fraunhofer IESE

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

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

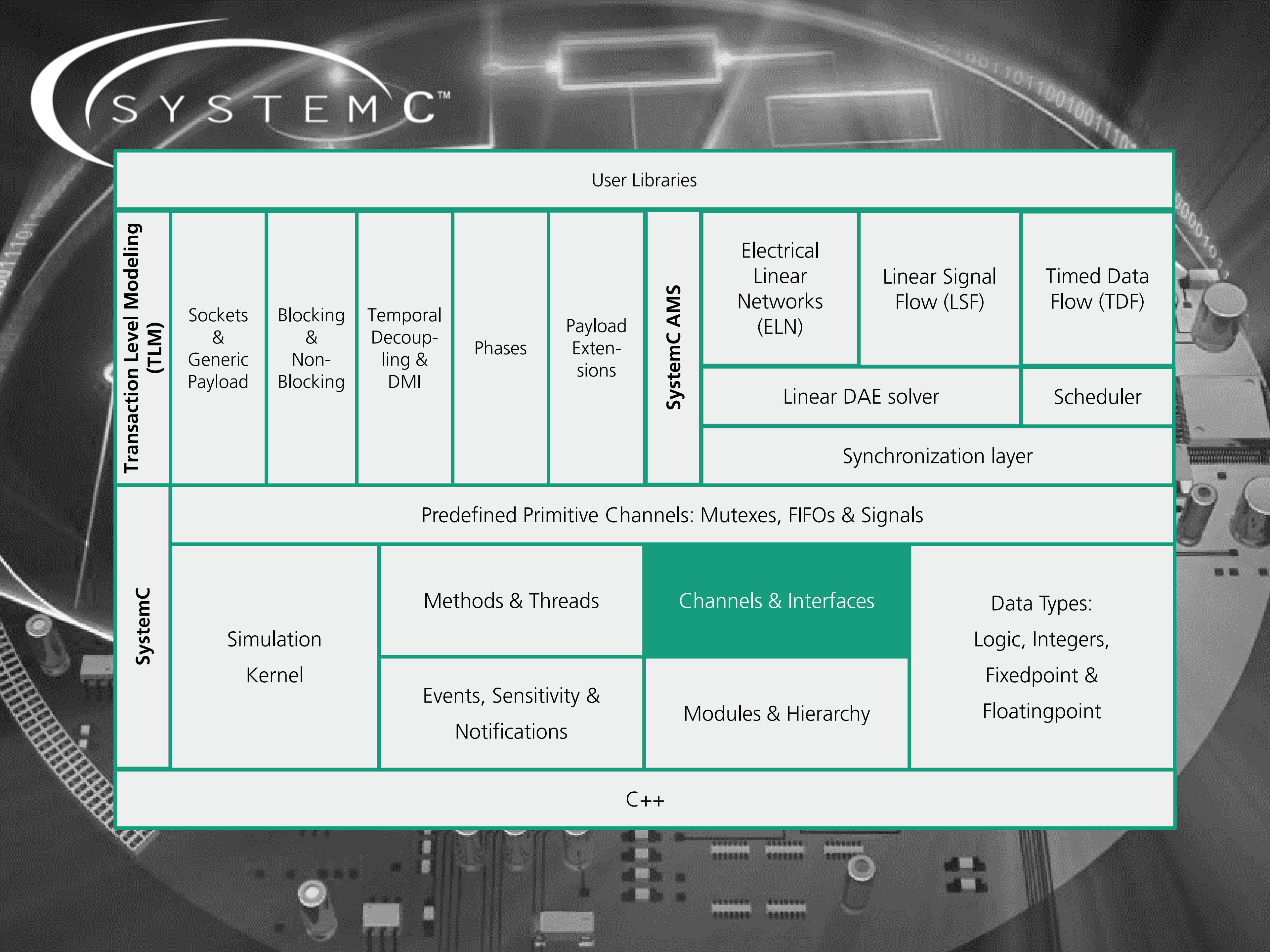
Why Virtual Base Class Concept for Channels?

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

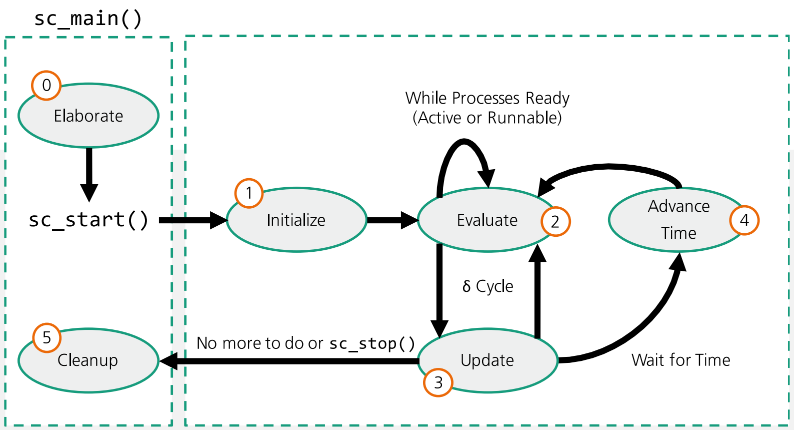
```
class memorySimple: public memoryInterface {
    public:
    void write(unsigned int addr, int data)
    {
        mem[addr] = data;
    }
    void int read(unsigned int addr)
    {
        return mem[addr];
    }
    private:
    int mem[1024];
}
```

```
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


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++

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;
};
```

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);
        }
    }        so we call sensitive<< we call this

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

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

Fraunhofer IESE

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;
}
```

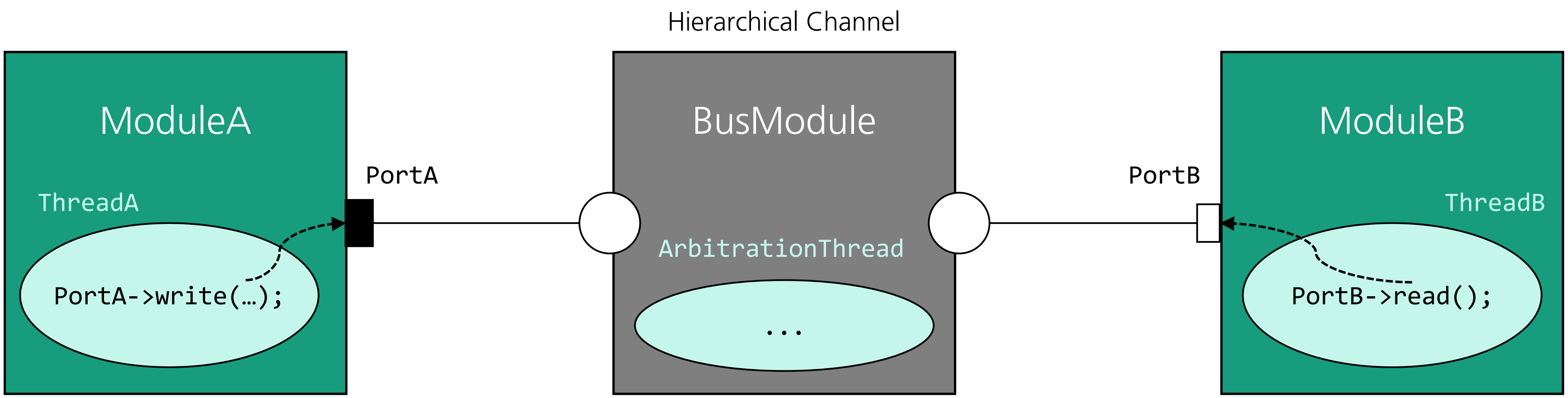
Try code on github:

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

Fraunhofer IESE

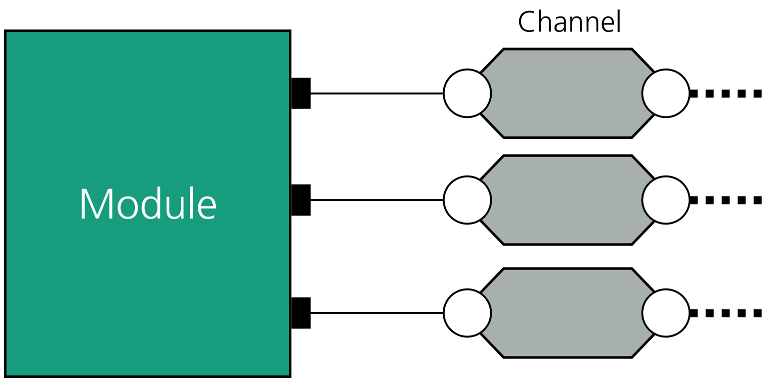
Hierarchical Channels

- 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"

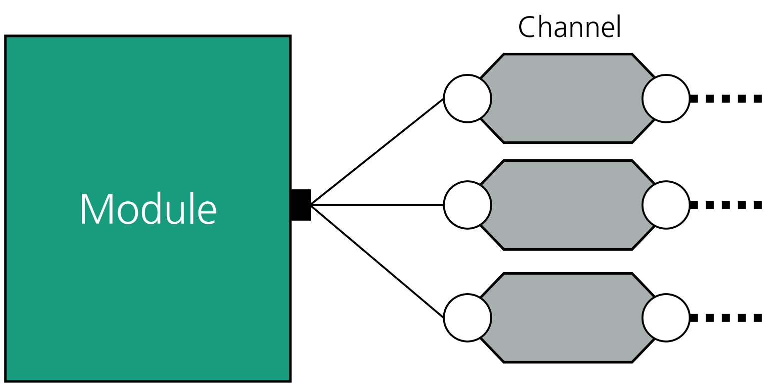


Fraunhofer IESE

Ports, Port Arrays and Multiports



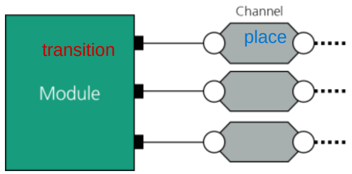
- Static declaration of ports and binding to separated channels.
- Is fixed on compile time
- Port arrays are more convenient



- Dynamic port creation during elaboration phase
- Using Multiports

Fraunhofer IESE

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);
        }
    }
};
```

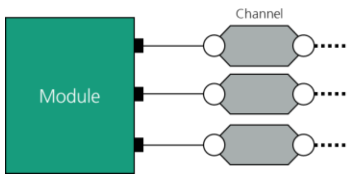
```
int sc_main(…)
{
    module m("m");
    sc_fifo<int> f1, f2, f3;

    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

Templated Port Arrays



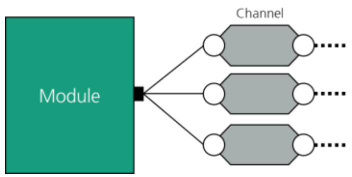

```
template <int N=1>
SC_MODULE(module
{
    sc_port<sc_fifo_out_if<int> > port[N];

    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);
        }
    }
};
```

```
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;
}
```

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

Fraunhofer IESE

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!

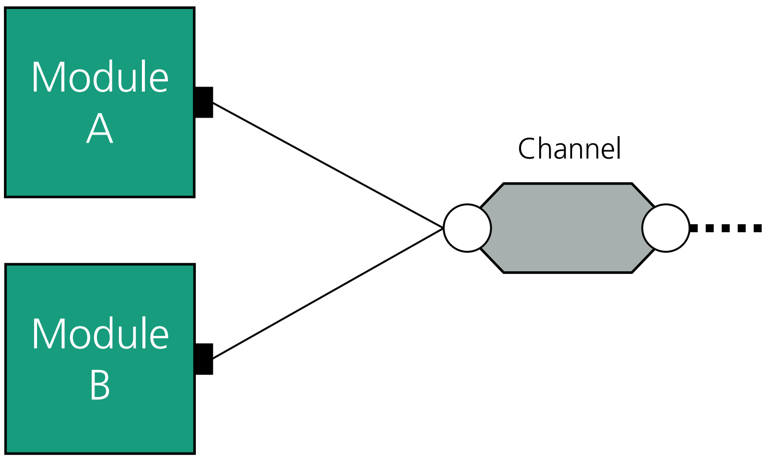
Try code on github:

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

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

Fraunhofer IESE

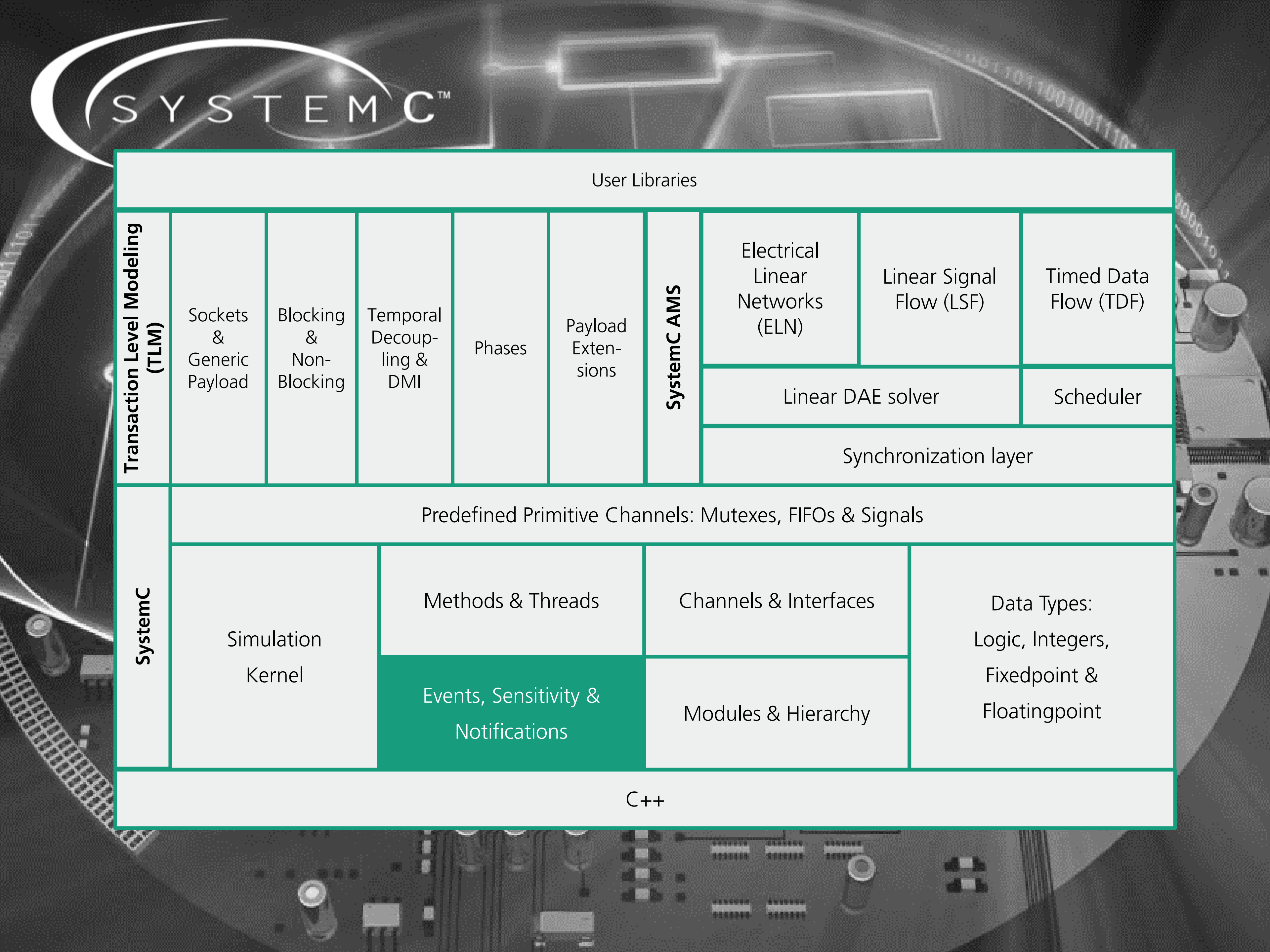
Multiple Bindings




```
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


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoupling & DMI
Phases
Payload Extensions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++

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 <u>current</u> 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

SystemC Events: sc_event_queue

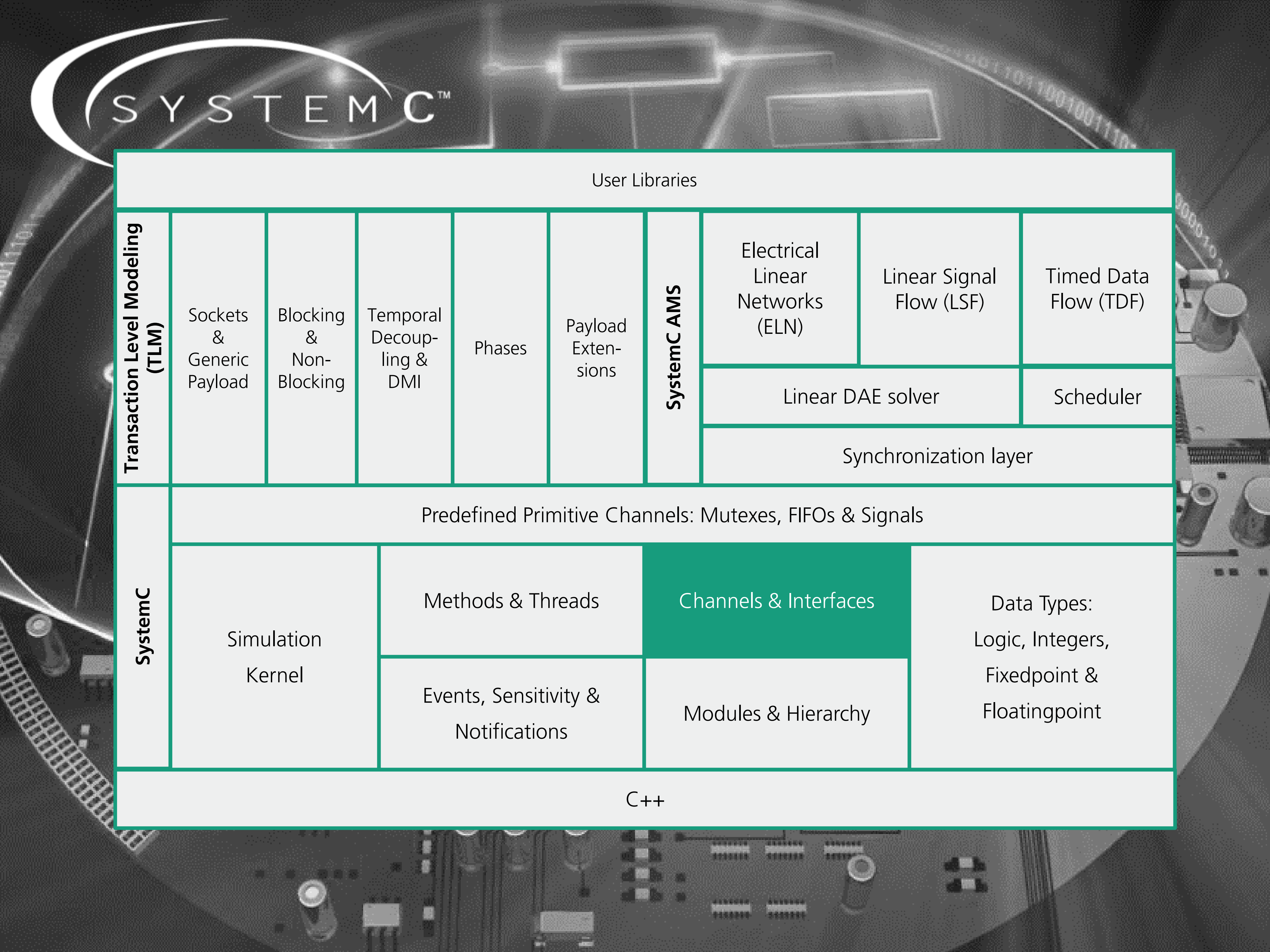
```
SC_MODULE(eventQueueTester) {
    sc_event_queue triggerEventQueue;

    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


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++

Event Finders

- Static Sensitivity:

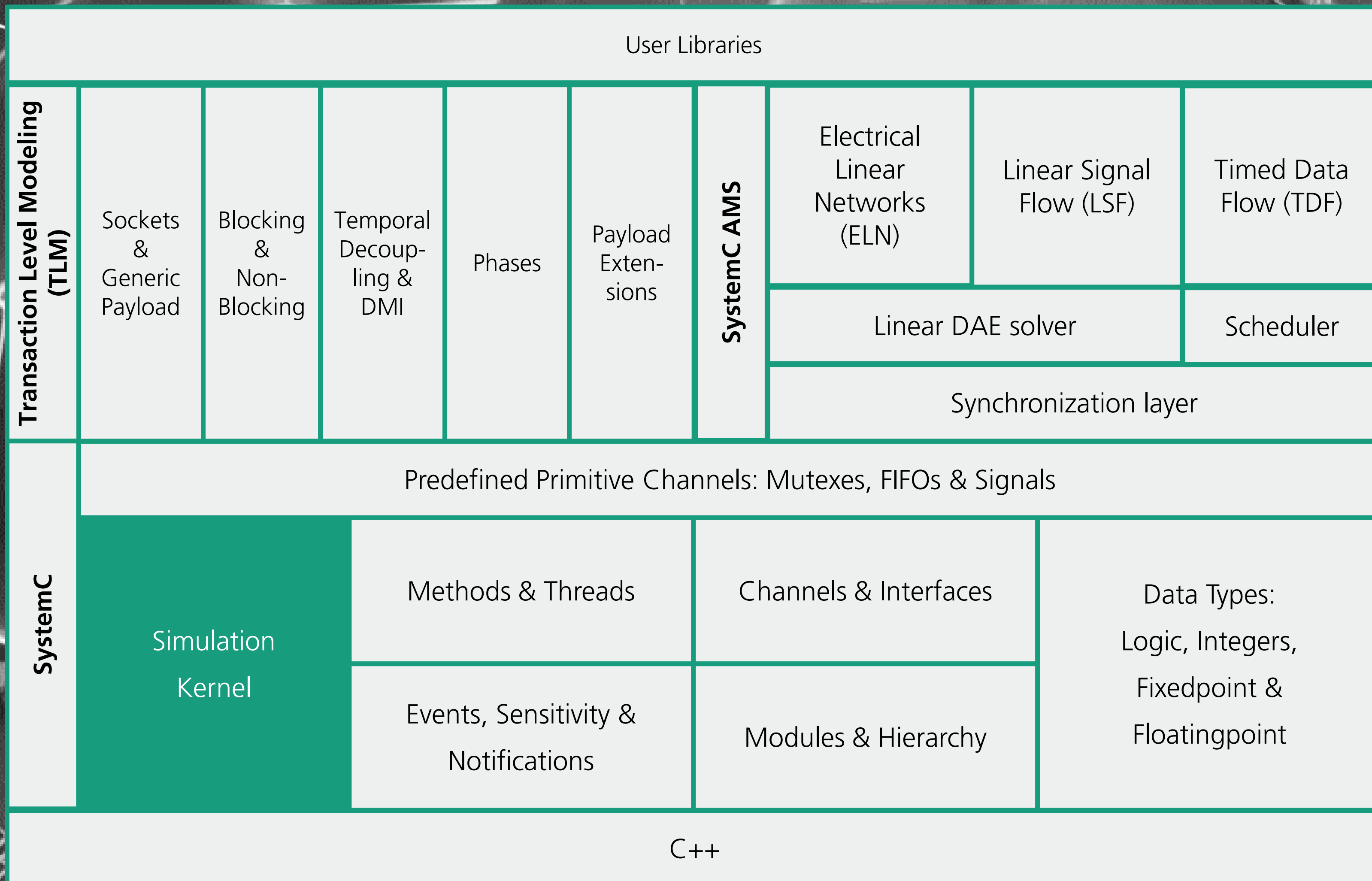
```
sensitive << clk;
sensitive << clk.default_event(); // Same
sensitive << clk.pos(); // Finding Special Event
```

- Finding special events is difficult because events are not part of the interface:
  - During runtime it is not clear, which channel will be bound and which events are implemented in the channel
- Solution:
  - Specialized Ports
  - Event Finders (`sc_event_finder`)

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

Fraunhofer IESE

SYSTEMC™


User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC
Predefined Primitive Channels: Mutexes, FIFOs & Signals
Simulation Kernel
Methods & Threads
Channels & Interfaces
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
Events, Sensitivity & Notifications
Modules & Hierarchy
C++

Dynamic Processes

- So far processes (threads and methods) were created during elaboration
- SystemC allows to generate new *dynamic* processes during simulation
- Fields of applications:
  - Testbenches
  - Verification
  - Modeling of SW
  - Modeling of OS
- 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!

Dynamic Processes

```
#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);
    }
};
```

```
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

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

Fraunhofer IESE

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`

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

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;
}
```

```
        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
```

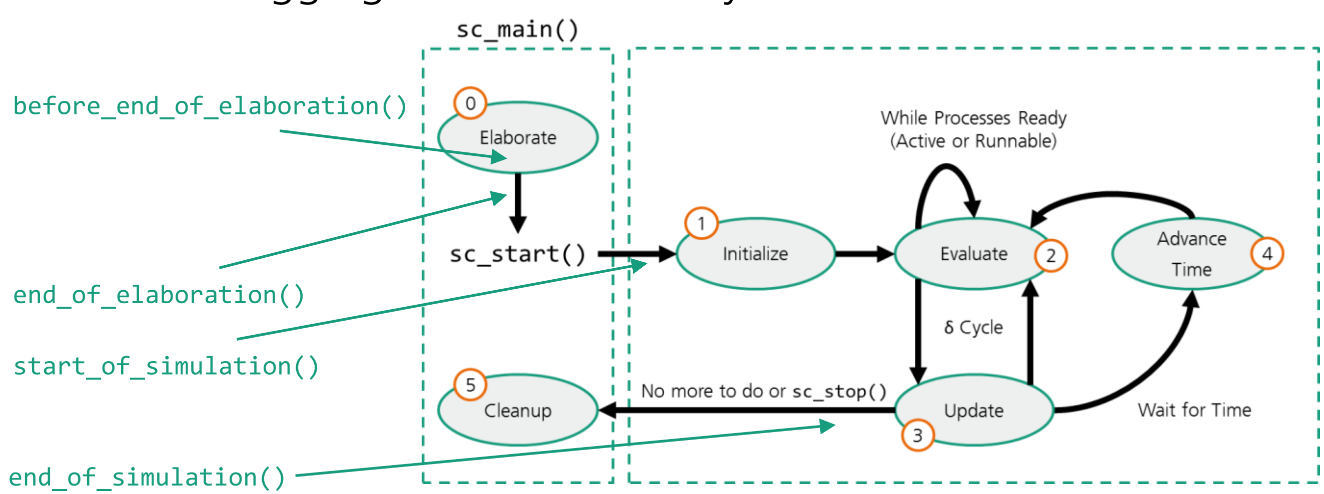
- Use custom reporthandler
- For more application/simulation specific output

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

Fraunhofer IESE

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

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.

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.

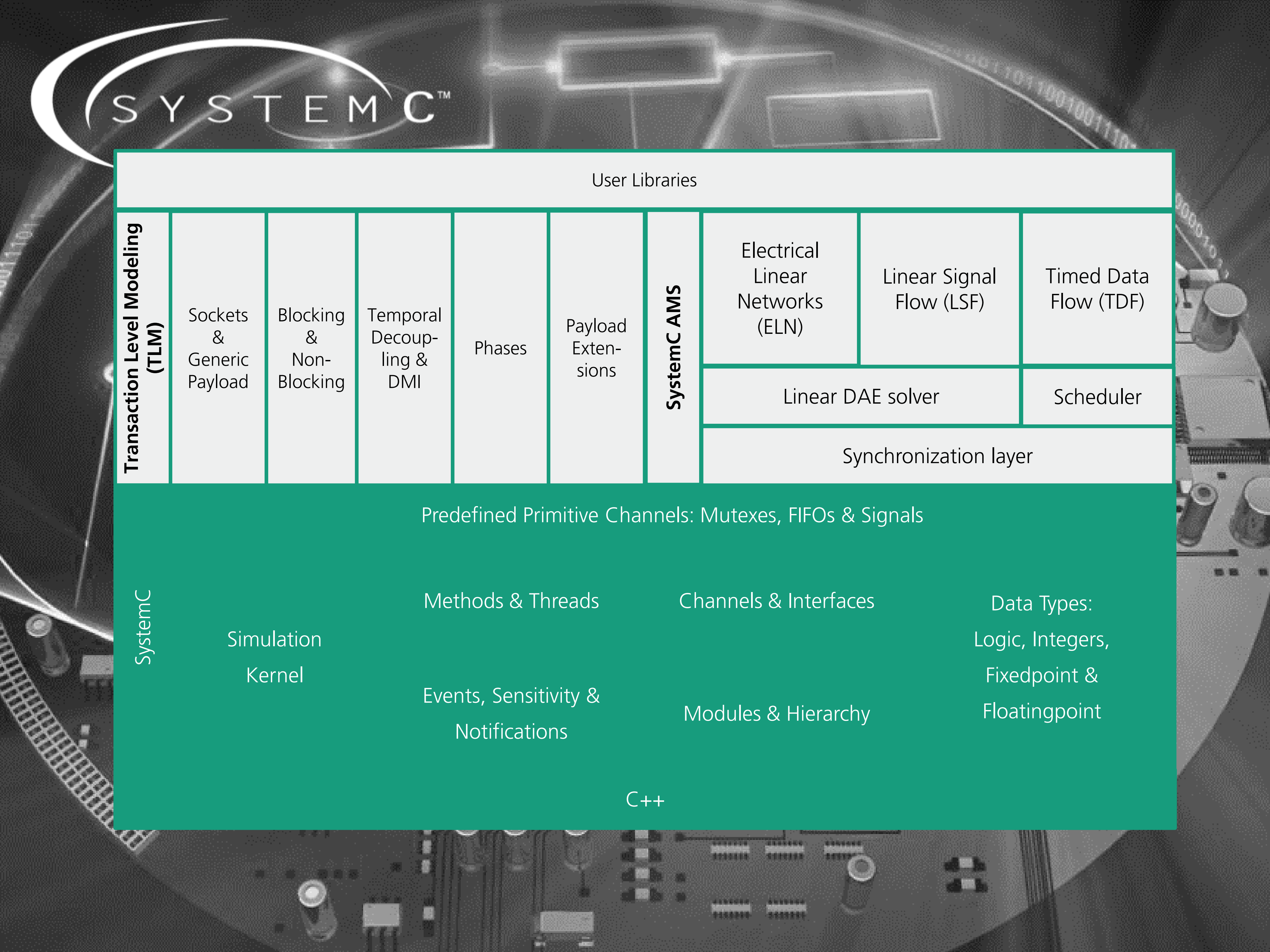
Callbacks

```
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


SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
Predefined Primitive Channels: Mutexes, FIFOs & Signals
SystemC
Simulation Kernel
Methods & Threads
Events, Sensitivity & Notifications
Channels & Interfaces
Modules & Hierarchy
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
C++