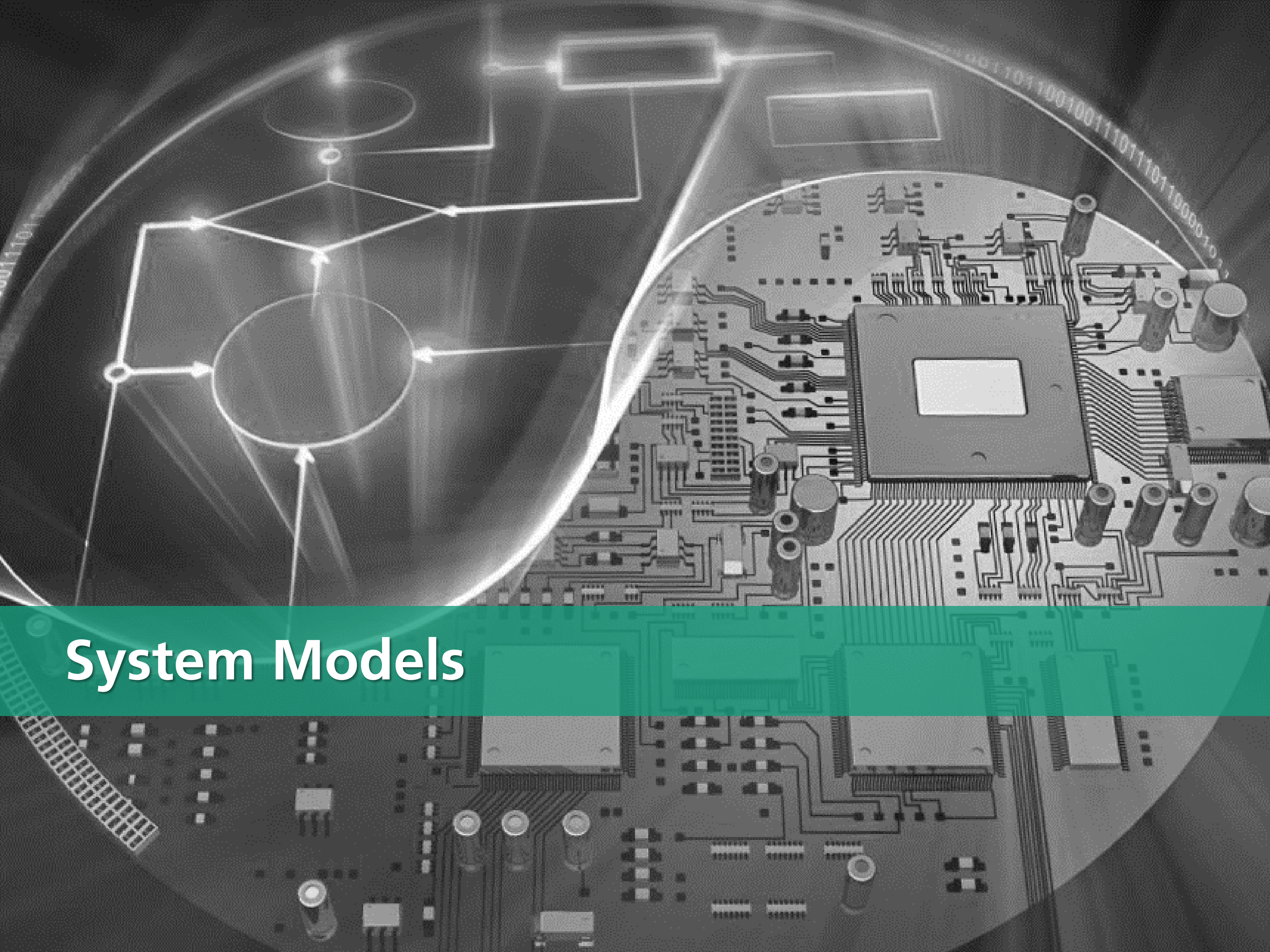
SystemC and Virtual Prototyping

Dr. Matthias Jung, Fraunhofer Institute IESE

matthias.jung@iese.fraunhofer.de

Fraunhofer
IESE



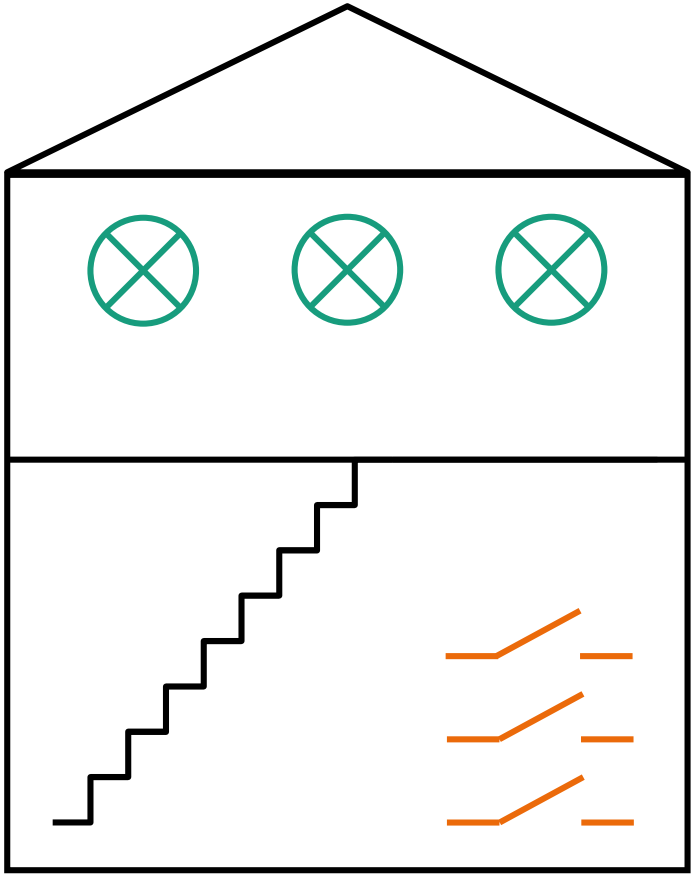
System Models

Test:

Models …

… enable … thinking

… hamper … perception



Fraunhofer IESE

Perception vs. Reality

What we perceive and how we interpret it depend on the frame through which we view the world around us.

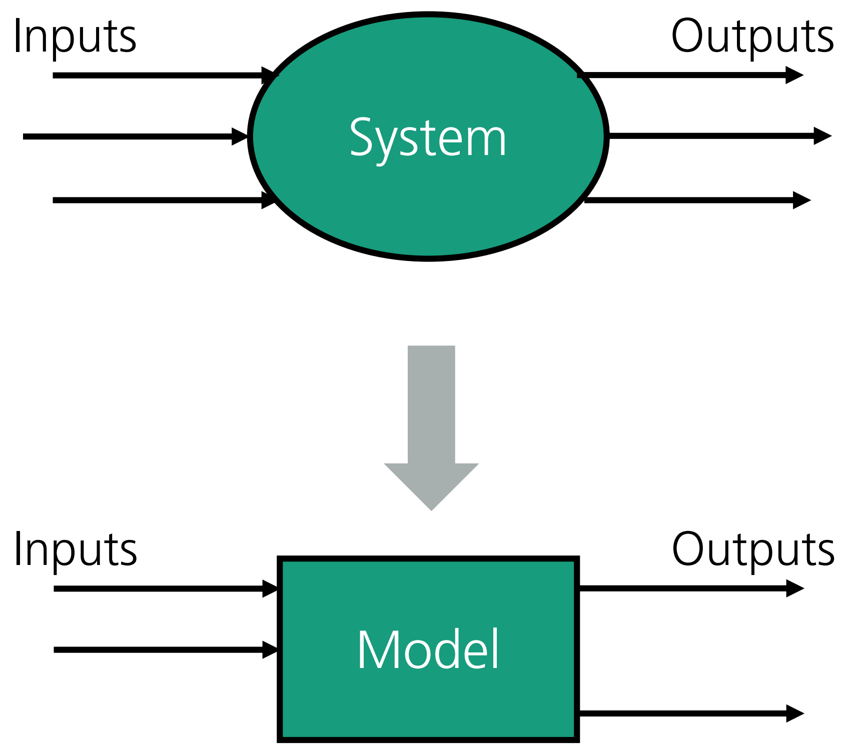
System and Model

- A **system** is a combination of components that act together to perform a function not possible with any of the individual parts

  Architecture describes how the system has to be implemented

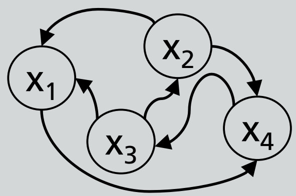
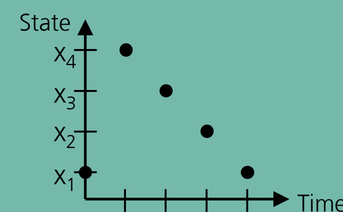
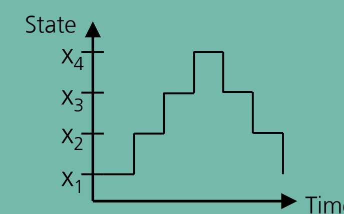
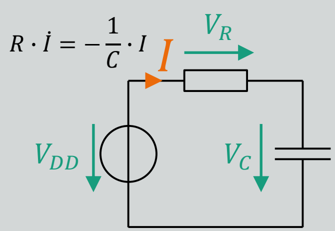
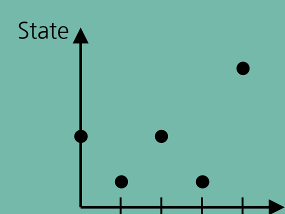
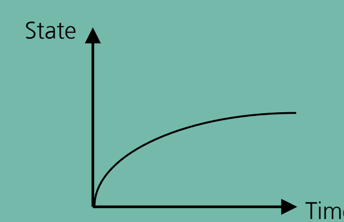
- A **model** is a formal description of the system, which covers selected information.

  Describes how the system works

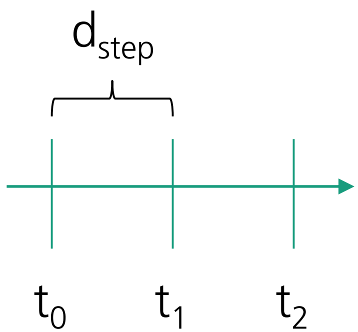


Fraunhofer IESE

Time and States

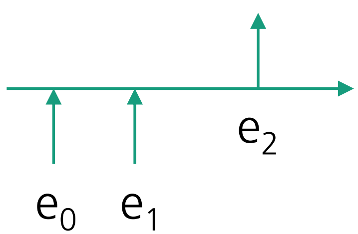
	Discrete Time Time values are countable (ℕ)	**Continuous Time** Time values are real (ℝ)
Discrete State State is countable (ℕ) 		
Continuous State States are real (ℝ) 		

Time Simulation Concepts



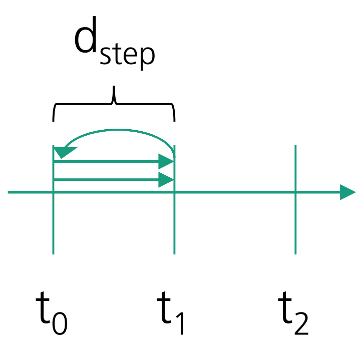
Discrete Time Simulation:

- Execution in fixed or variable discrete timesteps (d_{step})
- Input ports are constant during timesteps
- Output ports are updated at the end of a step
- Trade-Off between speed and accuracy



Discrete Event Simulation (DES)

- Events may occur at any time
- Events are sorted in a queue according to expiration time
- DES uses a two-dimensional time (superdense, delta delay)



Continuous Time Simulation

- Approximate the continuous behavior of physics -> diff. eq.
- Usually discrete timesteps are used
- Solver is used for simulation:
  - Errors are minimized by iterating the same simulation step

Fraunhofer IESE

MOC Support in SystemC

- **Discrete Event** as used for:
  - RTL Hardware Modeling
  - State Machines
  - Network Modeling (e.g. stochastic or "waiting room" models)
  - Transaction Level Modeling
- Continous Time with AMS-Extension
- Kahn Process Networks
- Static Multi-rate Data-flow
- Dynamic Multi-rate Data-flow
- Communicating Sequential Processes
- Petri Nets
- …

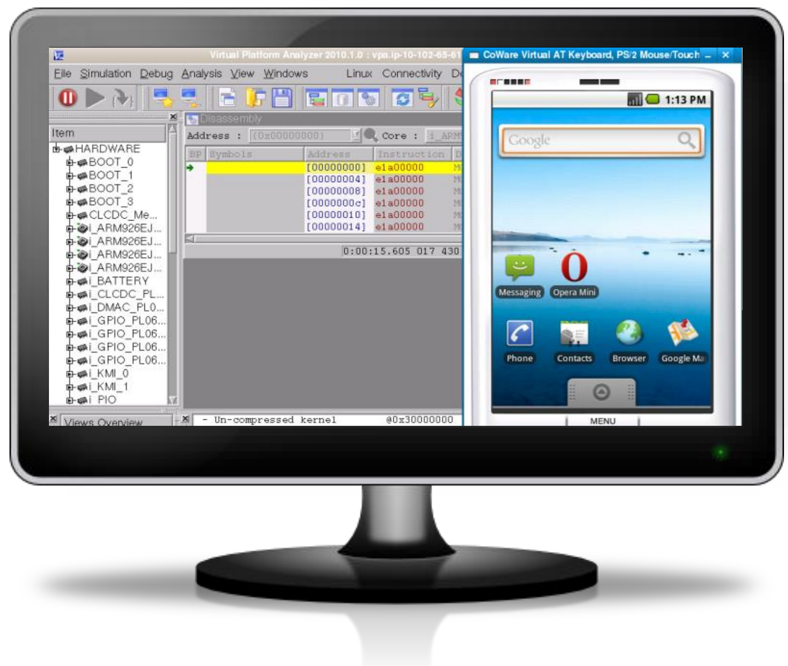
Wikipedia:

SystemC is a set of C++ classes and macros which provide an event-driven simulation interface (see also discrete event simulation). These facilities enable a designer to simulate concurrent processes, each described using plain C++ syntax.

SystemC Basics

Move to Virtuality?

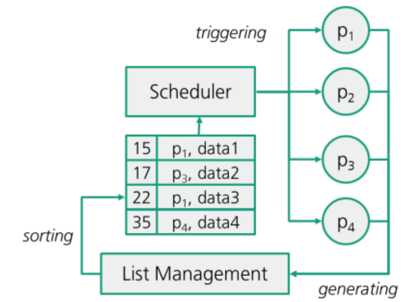
Everything is in the Developer's Desktop



Fraunhofer IESE

What is SystemC?

- Simulation and Modeling ~~Language~~ Library for C++
  - Discrete Event Model
  - IEEE Standard 1666 language for system-level-design
  - For complex systems consisting of hardware and software
  - Hardware / software co-design and co-simulation
  - Extension of hardware description languages to higher abstraction levels i.e. different levels of accuracy.



- Provides:
  - Set of library routines and macros implemented in C++ (class library)
  - Modeling concurrency
  - Synchronization
  - Inter-process communication
  - Simulation Kernel (scheduler) included
  - Compiler for C++ is sufficient for simulating SystemC models → binary is generated for executable simulation model

Fraunhofer IESE

Install SystemC on your Private Machine

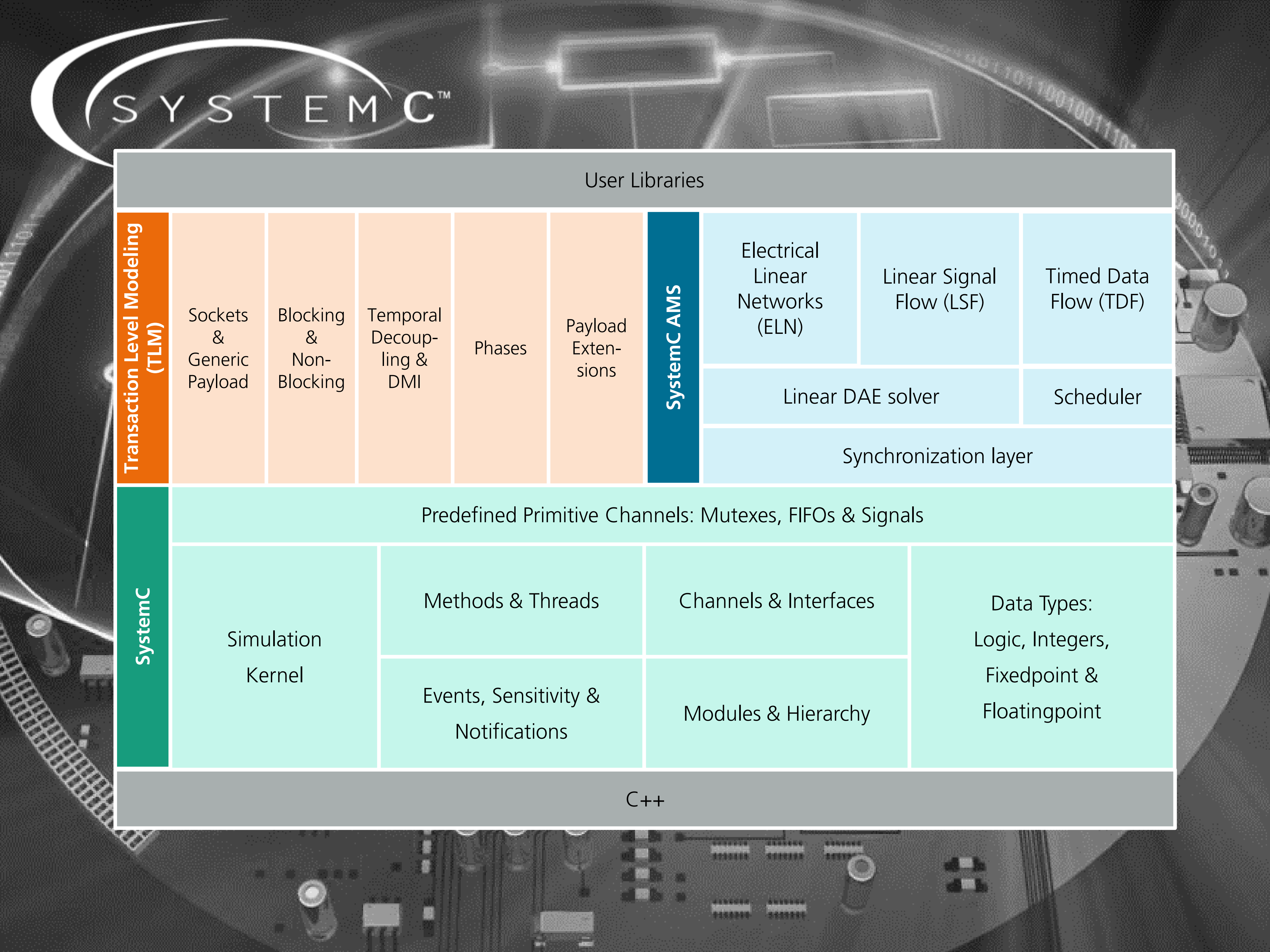
For Example on Ubuntu or Debian like Linux distributions

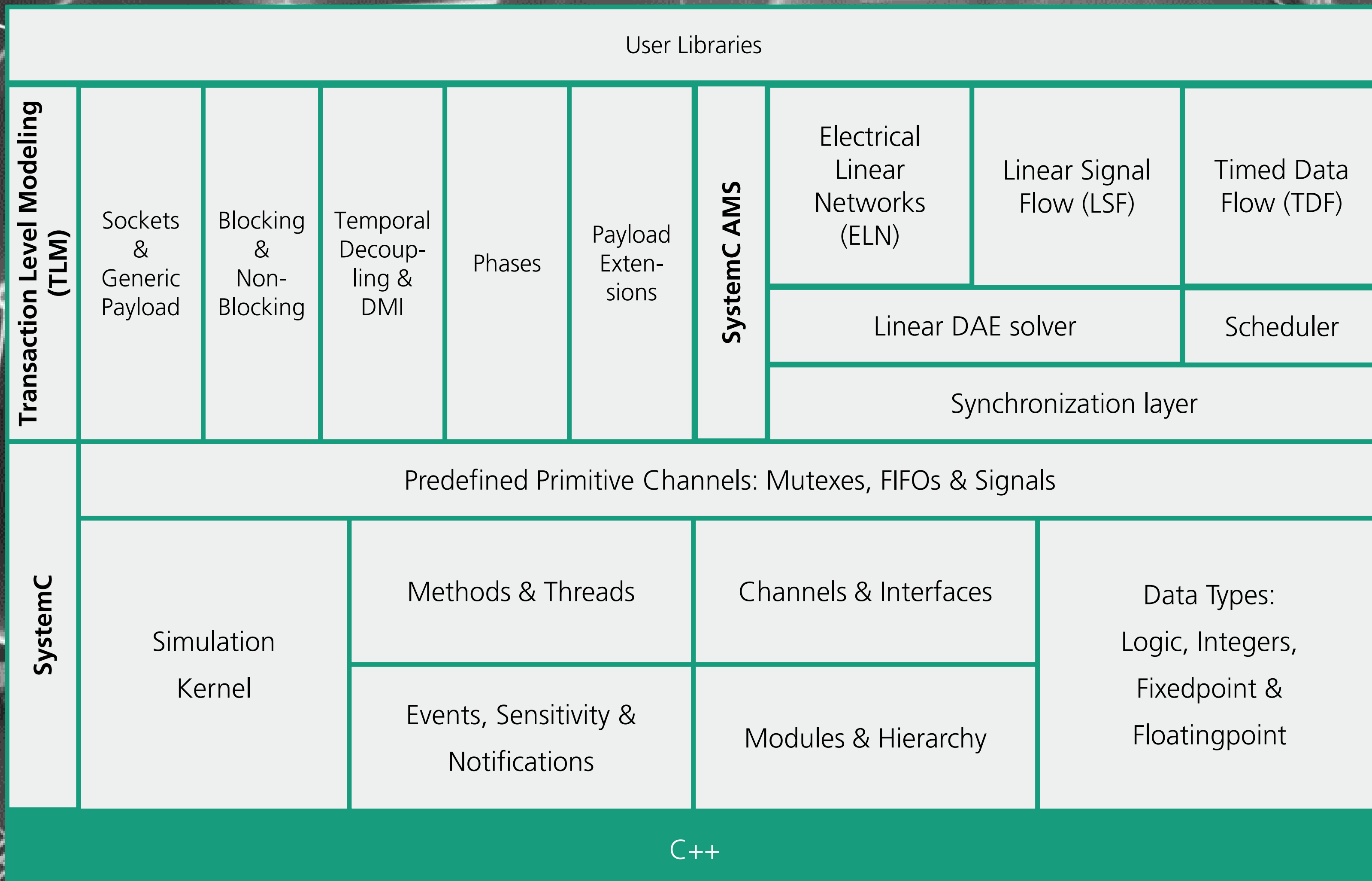
```
$ wget http://www.accellera.org/images/downloads/standards/systemc/systemc-2.3.1a.tar.gz
$ tar xfv systemc-2.3.1a.tar.gz
$ cd systemc-2.3.1a
$ ./configure --prefix=/opt/systemc/
$ make -j 4
$ sudo make install
```

Get script on GitHub:

https://github.com/tukl-msd/SCVP.artifacts/blob/master/install_systemc.sh

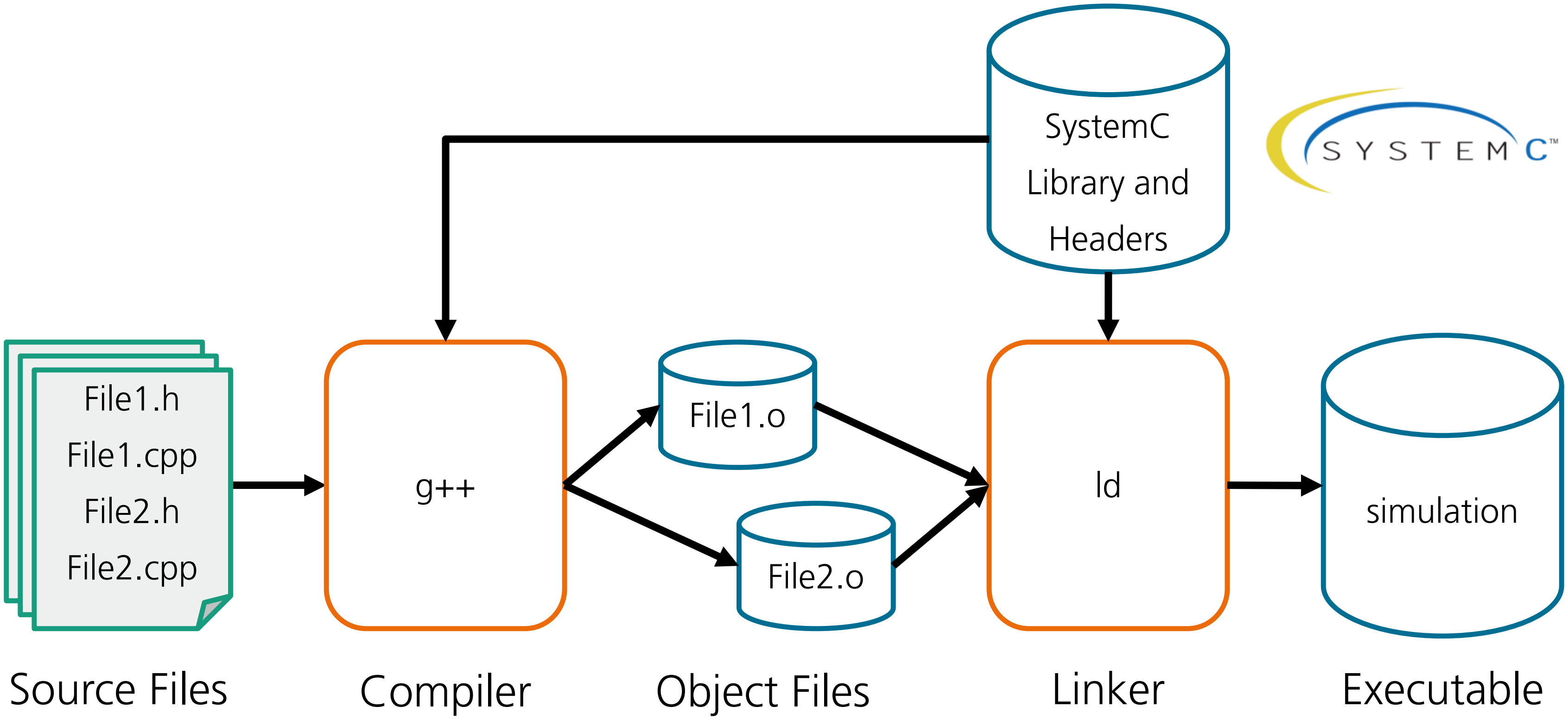
Fraunhofer IESE

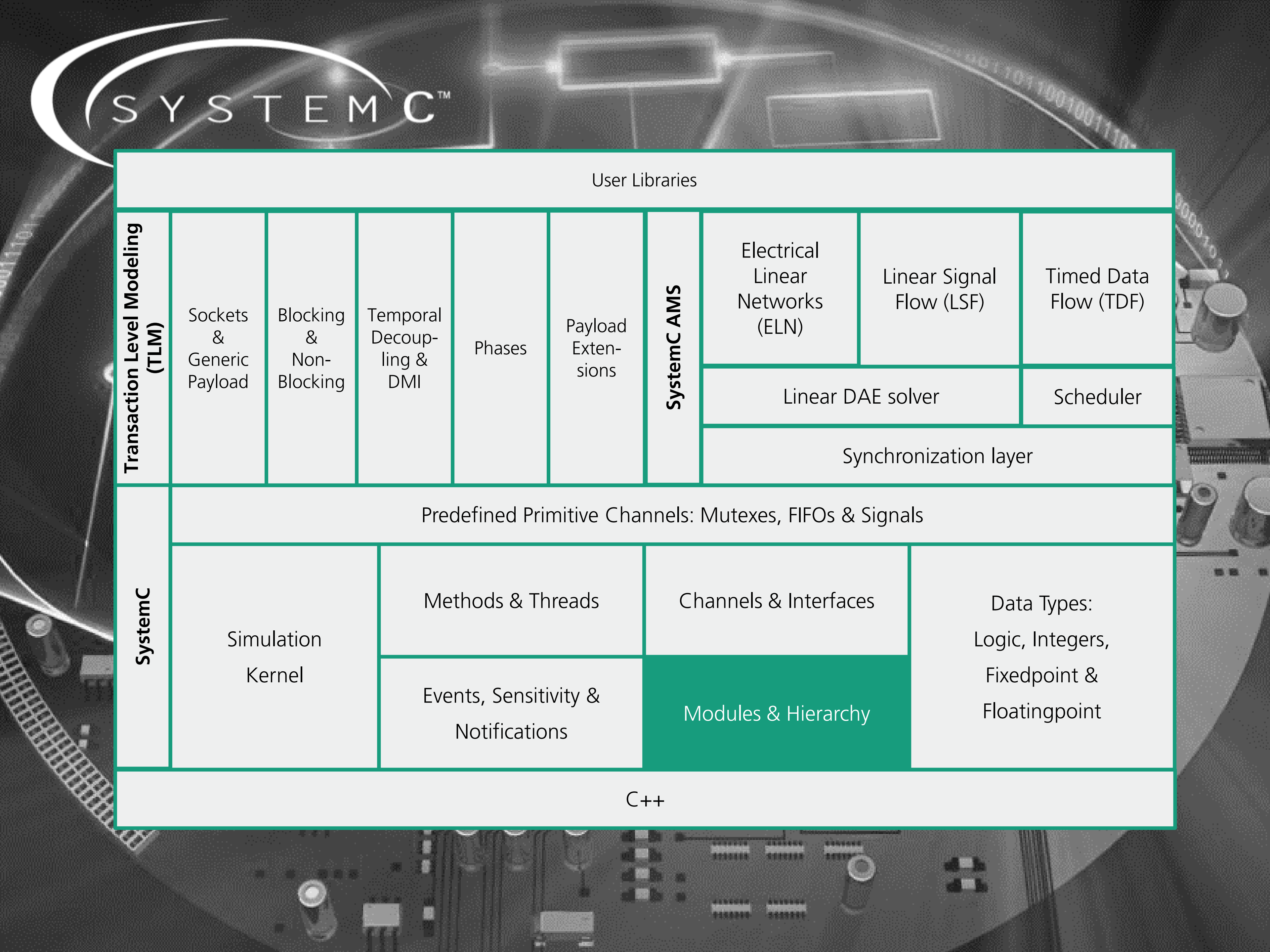

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

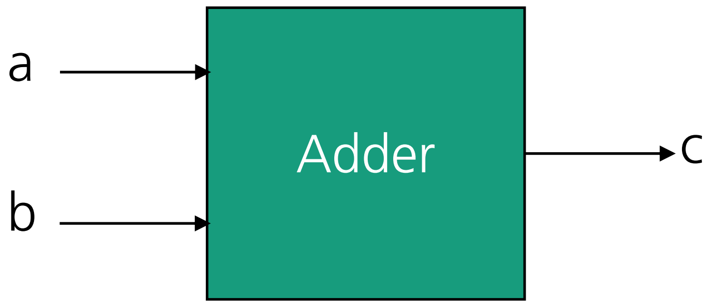
SystemC Compilation Flow

- SystemC is not a „language“!
- It’s just a set of classes and macros in a a C++ library




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++

SystemC Basic Example



Remember an adder in VHDL:

```
entity adder is
Port (
        a: in unsigned;
        b: in unsigned;
        c: out unsigned
);
end adder;

architecture arch of adder is
    adding: process (a,b)
        c = a + b;
    end process adding;
end arch;
```

SystemC Basic Example

```
SC_MODULE (adder)
{
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> c;

    void compute()
    {
        c.write(a.read() + b.read());
    }

    SC_CTOR (adder)
    {
        SC_METHOD (compute);
        sensitive << a << b;
    }
};
```

Module declaration

Define module input port named "**a**" with data type int

Implement functionality in member function **compute()**

Module constructor

Register function **compute()** at the SystemC scheduler as process

Tell the scheduler that **compute()** is sensitive to the input ports a and b

Fraunhofer IESE

SC_MODULE and SC_CTOR Macros

- `SC_MODULE(XYZ)` is a short macro for:

```
class XYZ : public sc_module
```

- `SC_CTOR(XYZ)` is a short macro for:

```
SC_HASPROCESS(XYZ);
XYZ(const sc_module_name &name) : sc_module(name)
```

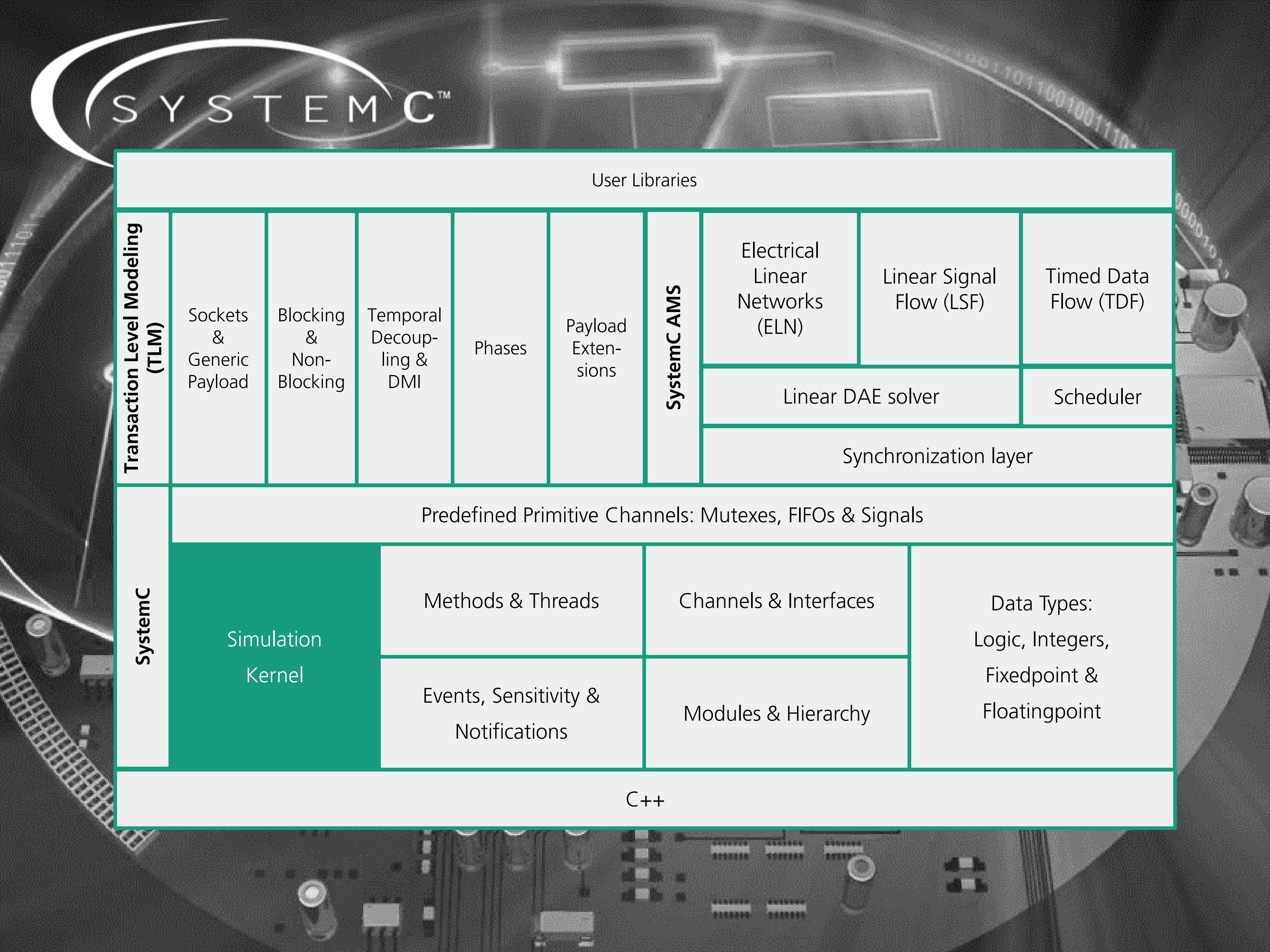
- `SC_HASPROCESS(XYZ)` is a short macro for:

```
typedef XYZ SC_CURRENT_USER_MODULE
```

If you want to have constructor arguments for your SystemC module it is preferable not to use `SC_CTOR`, declare the normal constructor and use the `SC_HASPROCESS` instread.

- Not be confused with a process, its just that SystemC needs the class name for internal declarations for example in `SC_METHOD` or `SC_THREAD`. SystemC cannot know beforehand how you will call your module.
(What is typedef?: `typedef unsinged long ul;`)

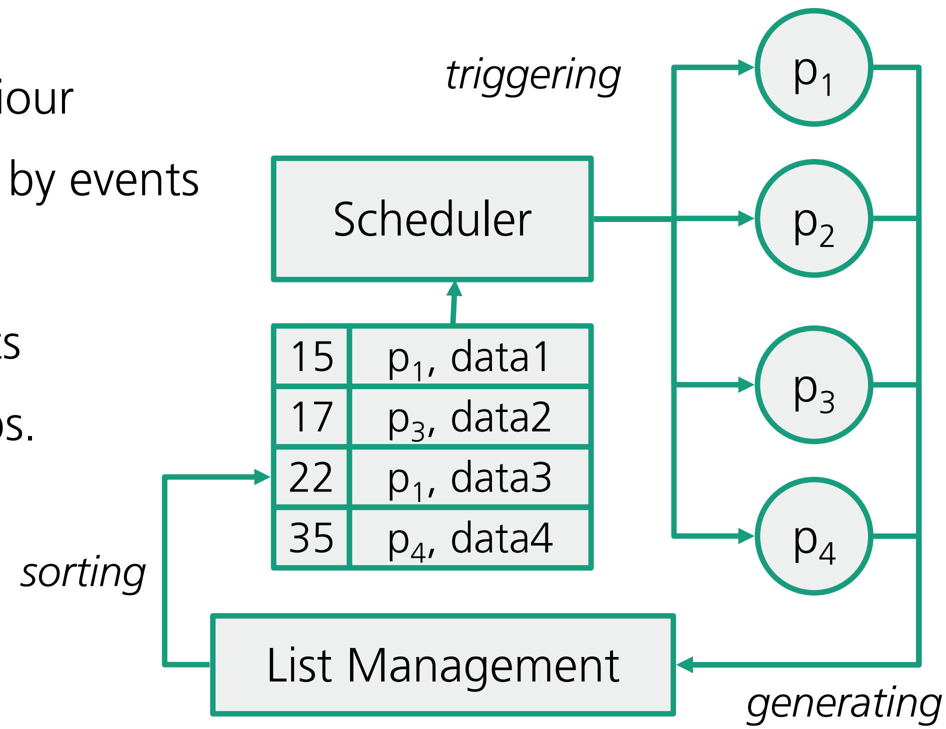
Fraunhofer IESE


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++

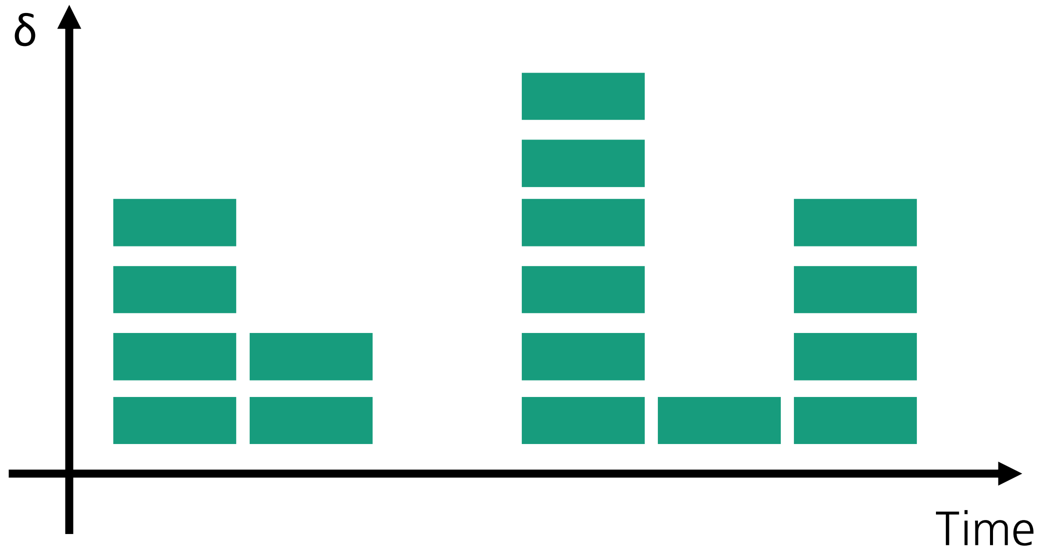
Discrete Event Models (DEM) – General Concept

Evaluation of state changes only at occurrence of events!

- Process describes functional behaviour
- Execution of processes is triggered by events
- Processes are deterministic
- Processes may generate new events
- Events are sorted w.r.t. time stamps.

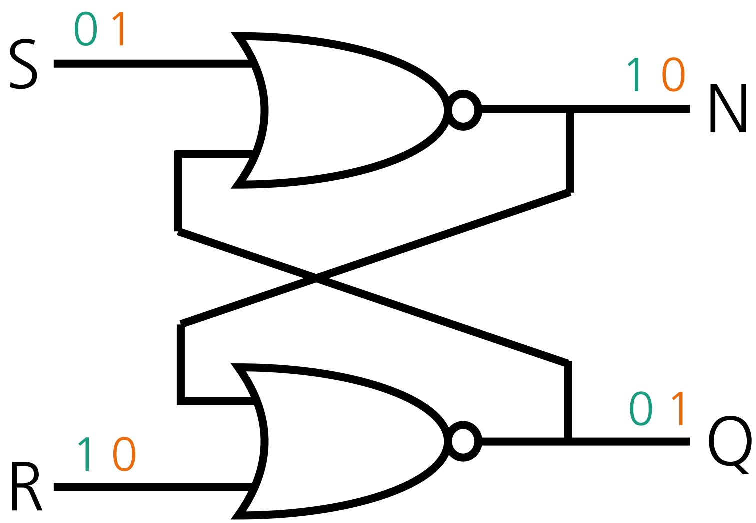


The δ-Delay – A Concept of a Two-Dimensional Time



- The δ-Delay enables the simulation of concurrency in a sequential simulator
- The δ-Delay is an infinitesimally small abstract time unit
- The δ-Delay guarantees a deterministic signal assignment
- The δ-Delay is used, if a statement with 0 ns or SC_ZERO_TIME is called.

Example for δ-Cycles: RS-Latch



	Time	S	R	Q	N
S=0, R=1	0 ns + 0δ	0	1	0	1
@10ns S=1, R=0	10 ns + 0δ	1	0	0	1
	10 ns + 1δ	1	0	0	0
	10 ns + 2δ	1	0	1	0
	10 ns + 3δ	1	0	1	0

- The functionality of the RS Latch is modelled by a process P:

$$Q^* = \overline{R \vee N}$$

$$N^* = \overline{S \vee Q}$$

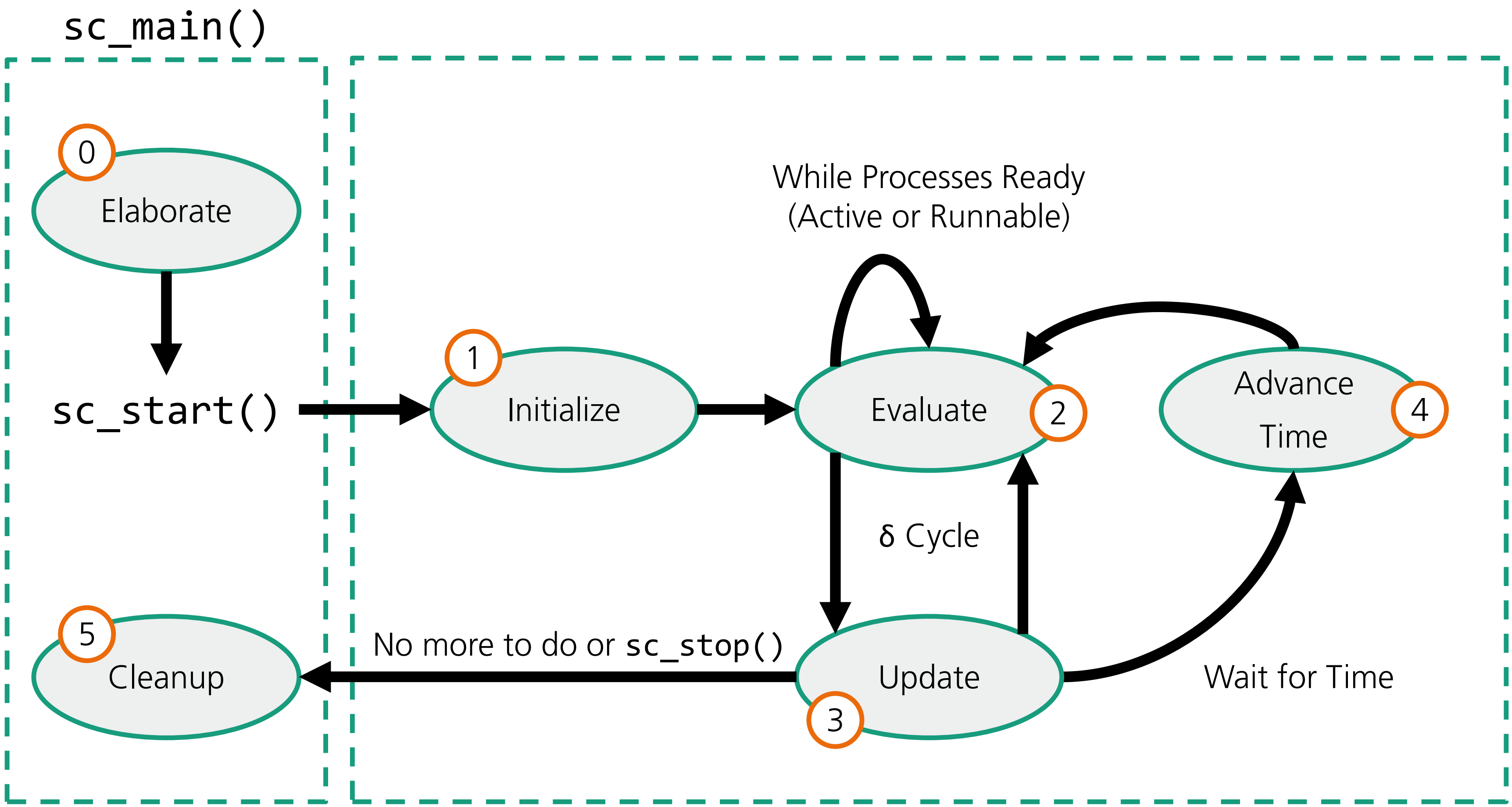
- P is sensitive to events (i.e. signal changes of S, R, N and Q)
- The output of Q depends on N
- The output of N depends on Q

Try code on github:

https://github.com/tukl-msd/SCVP.artifacts/tree/master/delta_delay

Fraunhofer IESE

SystemC Simulation Kernel



Fraunhofer IESE

SystemC Simulation Kernel

0 ***Elaborate***: Execution of all states prior to the `sc_start()` call are known as the elaboration phase. All constructors of all `SC_MODULE`s are called, the connections (bindings) between the different modules is checked. If for example a port is not bound the simulation will complain here in the beginning.

1 ***Initialize:*** During Initialization, each process is executed once (for `SC_METHOD`) or until a synchronization point (i.e. `wait()`) is reached (for `SC_THREAD`). In some circumstances it may not be desired for all processes to be executed in this phase. To turn off initialization for a process, we may call `dont_initialize()` after its `SC_METHOD` or `SC_THREAD` declaration inside the constructor. The order in which these processes are executed is unspecified, however, it is deterministic (for every simulation run with the same SystemC version it will behave the same way).

Fraunhofer IESE

SystemC Simulation Kernel

2 ***Evaluate***: From the set of processes marked as executable, all processes are executed successively and in an undefined order, and the marking is removed. An `SC_METHOD` is executed until the return, an `SC_THREAD` is suspended by calling a `wait(…)` statement. A process can not be interrupted during execution. By writing to `sc_signals` or `sc_fifos` etc., so-called update requests will be created in this phase for assignments to be made in the update phase 3. These update requests are noted by the scheduler. Furthermore, the execution of a `wait(…)` may result in a "timeout". This means that this process should be continued at a later time and they are stored in the event queue.

```
mySignal.write(true);
```

```
template< class T, sc_writer_policy POL > inline void
sc_signal<T,POL>::write( const T& value_ ) {
    bool value_changed = !( m_cur_val == value_ );
    [...]
    m_new_val = value_;
    if( value_changed ) {
        request_update();
    }
}
```

A look into the SystemC Kernel

Fraunhofer IESE

SystemC Simulation Kernel

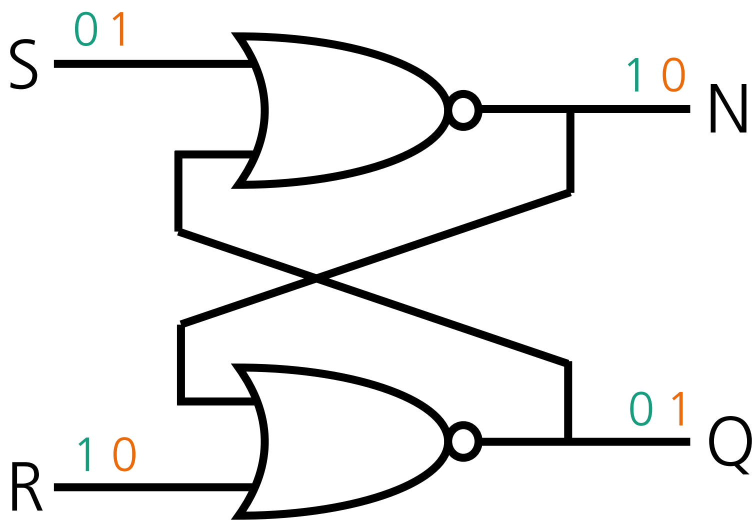
3 ***Update***: In this phase, the previously requested updates are performed. The scheduler estimates if processes are sensitive to updates of these signals and mark them as executable. Then the scheduler goes again to the evaluation phase 2 (This looping is called a δ Cycle). If there are no new processes marked for execution we proceed to 4

4 ***Advance Time***: Processes sensitive to events in the event queue with the smallest time are marked for execution and the scheduler proceeds to the evaluation phase 2 and thus, the simulation time is advanced. If there are no events in the event queue the simulation is finished. Then the scheduler proceeds to the cleanup phase 5 where all destructors are called.

- Note that calling `sc_stop()` in a process will directly lead to phase 5.

Fraunhofer IESE

<u>Remember:</u> Example for Delta Delay: RS-Latch



	Time	S	R	Q	N
S=0, R=1	0 ns + 0δ	0	1	0	1
@10ns S=1, R=0	10 ns + 0δ	1	0	0	1
	10 ns + 1δ	1	0	0	0
	10 ns + 2δ	1	0	1	0
	10 ns + 3δ	1	0	1	0

In SystemC Code:

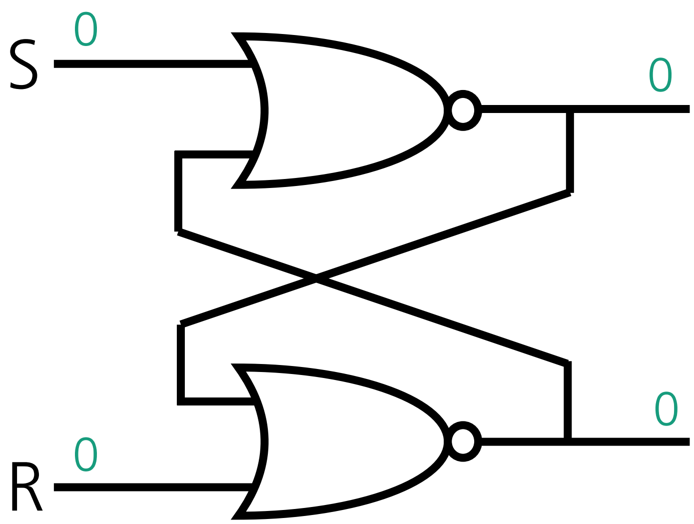
```
SC_MODULE(rslatch)
{
    sc_in<bool> S;
    sc_in<bool> R;
    sc_out<bool> Q;
    sc_out<bool> N;

    SC_CTOR(rslatch) : S("S"), R("R"), Q("Q"), N("N")
    {
        SC_METHOD(process);
        sensitive << S << R << Q << N;
    }

    void process()
    {
        Q.write(!(R.read()||N.read())); // NOR Gate
        N.write(!(S.read()||Q.read())); // NOR Gate
    }
};
```

Try code on github:

https://github.com/tukl-msd/SCVP.artifacts/tree/master/delta_delay

Fraunhofer IESE

Problem: Feedback Loops



Time	S	R	Q	N
0 ns + 0δ	0	0	0	0
0 ns + 0δ	0	0	1	1
0 ns + 1δ	0	0	0	0
0 ns + 2δ	0	0	1	1
0 ns + 3δ	0	0	0	0
…	…	…	…	…
0 ns + ∞δ	0	0	?	?

- In some rare occasions circuit can oscillate
- Infinite loop of δ-cycles – i.e. waiting forever
- Simulation time will never advance

Try code on github:

https://github.com/tukl-msd/SCVP.artifacts/tree/master/feedback_loop

Fraunhofer IESE

Order of Execution

Using Normal Variables:

```
void process() // int E=5 F=6
{
    E = F;
    F = E;
}
```

- Result is E = 6 and F = 6
- Swapping is not possible without a temporary variable

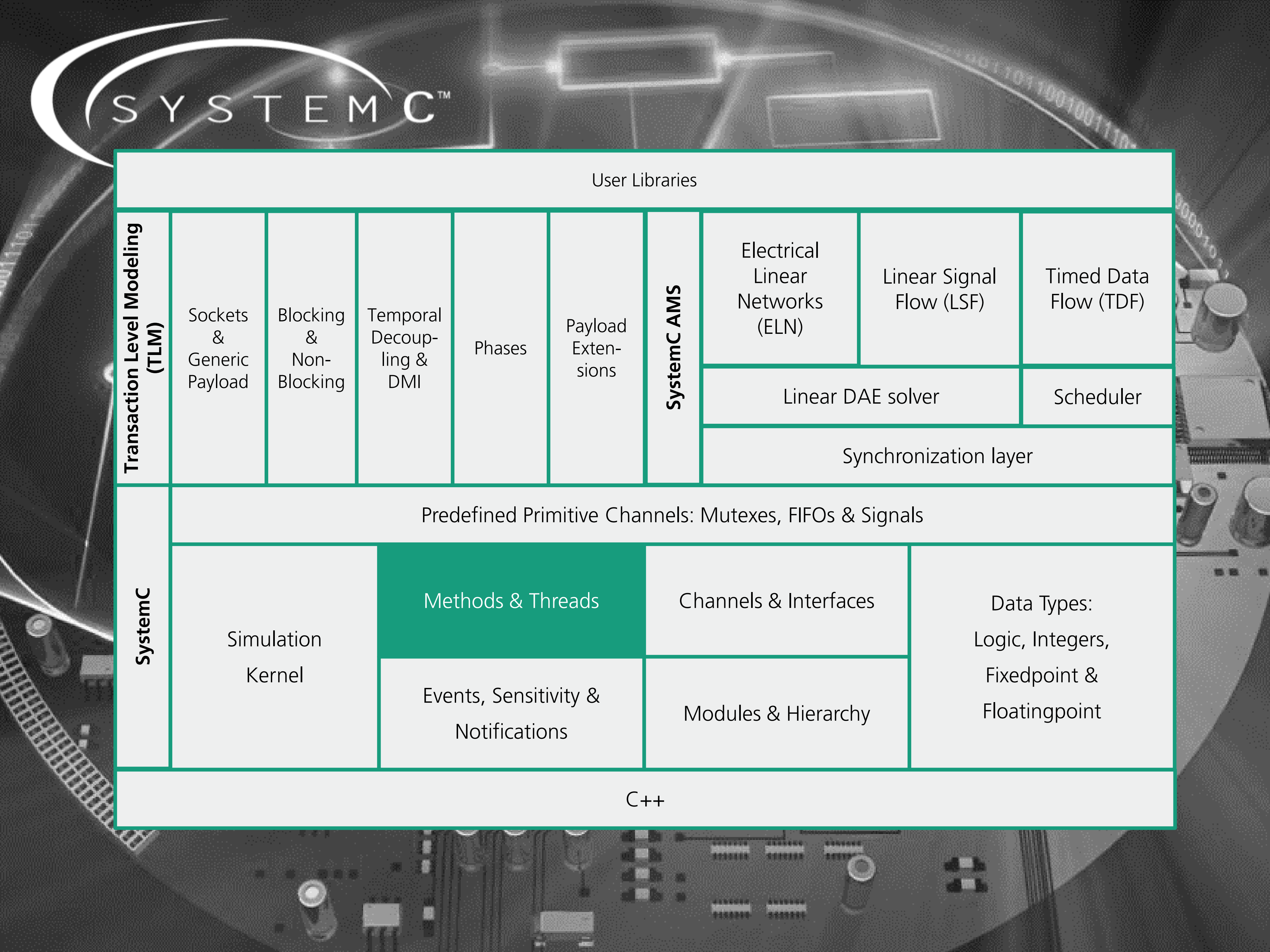
Using sc_signals etc.:

```
void process() // sc_signal<int> C=3 D=4
{
    C = D;
    D = C;
}
```

- Result is C = 4 and D = 3
- “Concurrent” execution of the statements

Try this as code on GitHub:

https://github.com/tukl-msd/SCVP.artifacts/tree/master/swapping_example


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++

Methods and Threads

In SystemC there exist two ways of representing processes, called:

Methods (SC_METHOD)

- Similar to Verilog's always and VHDL's process
- Atomic execution of method, with no preemption – i.e. complete scope is executed {}
- Therefore, infinite loops must be avoided
- Methods are usually sensitive to signals and events in the sensitivity list
- Methods can be called as often as possible – e.g. a signal change may trigger process again (δ cycle)

Threads (SC_THREAD)

- Threads are only started once at the begin of the simulation – i.e. if end of the scope is reached the thread dies.
- Threads can be suspended using the `wait(…)` statement
- Infinite loops are allowed and even needed
- Threads have much more overhead because of context switches
- Threads are good for test benches and TLM

SC_METHOD Example:

Example: the RS Latch

- A change on the S or R input triggers the method
- However, the method changes Q and N such that the method is again triggered in the next delta cycle
- This ‘fakes’ concurrency within the method

Try code on github:

https://github.com/tukl-msd/SCVP.artifacts/tree/master/delta_delay

```
SC_MODULE(rslatch)
{
    sc_in<bool> S;
    sc_in<bool> R;
    sc_out<bool> Q;
    sc_out<bool> N;

    SC_CTOR(rslatch) : S("S"), R("R"), Q("Q"), N("N")
    {
        SC_METHOD(process);
        sensitive << S << R << Q << N;
    }

    void process()
    {
        Q.write(!(R.read()||N.read())); // NOR Gate
        N.write(!(S.read()||Q.read())); // NOR Gate
    }
};
```

Each instance of an `sc_module` needs a name.

Fraunhofer IESE

SC_THREAD Example:

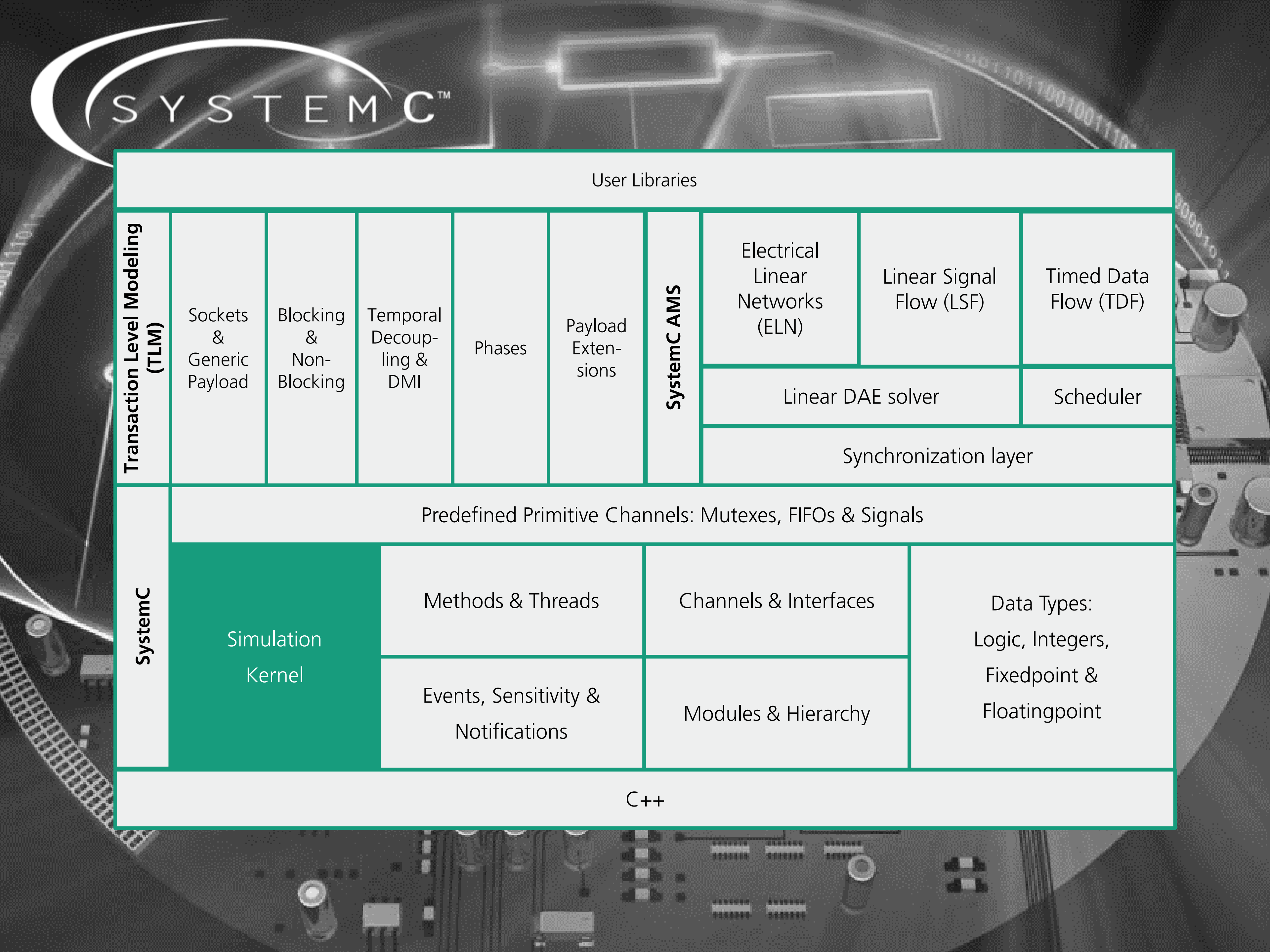
Example: the RS Latch

- For `SC_THREAD`s it is important that they have loops and `wait` statements otherwise they die.
- `SC_THREAD`s can be suspended by `wait` statements, `SC_METHOD`s can not!

Try code on github:

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

```
#include "systemc.h"

SC_MODULE(rslatch) {
    sc_in<bool> S;
    sc_in<bool> R;
    sc_out<bool> Q;
    sc_out<bool> N;

    SC_CTOR(rslatch) : S("S"), R("R"), Q("Q"), N("N") {
        SC_THREAD(process);
        sensitive << S << R << Q << N;
    }

    void process() {
        while(true) {
            wait();
            Q.write(!(R.read()||N.read())); // Nor Gate
            N.write(!(S.read()||Q.read())); // Nor Gate
        }
    }
};
```

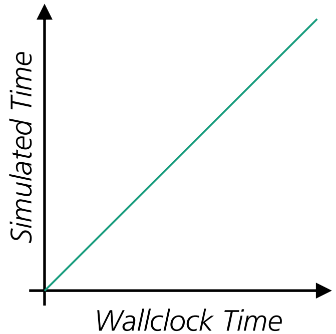

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
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++

Notion of Time in Simulations

- ***Wall-Clock Time:*** the time from the start of execution to completion of the simulation for a human observer.
- ***Simulated Time:*** is the time being modeled by the simulation which may be less than or greater than the simulation's wall-clock time.

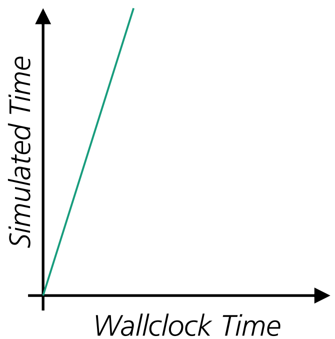
"Real-Time" (HiL)

WCT = ST

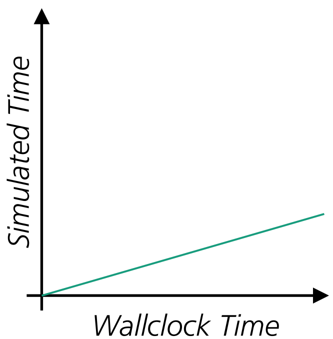


"As Fast as Possible" (vHiL)

WCT < ST



WCT > ST



Fraunhofer IESE

SystemC's Notion of Time

- ***Wall-Clock Time:*** the time from the start of execution to completion of the simulation for a human observer.
- ***Simulated Time:*** is the time being modeled by the simulation which may be less than or greater than the simulation's wall-clock time.

- SystemC tracks time with 64 bits of resolution using a class known as `sc_time`
- The global time is advanced within the kernel

Fraunhofer IESE

SystemC's Notion of Time

- `sc_time` is usually declared as: `sc_time name(`*`double`*`, `*`sc_time_unit`*`);`
- `sc_time` Provides all typical operands `+`, `-`, `*`, `/`, `==`, `!=`, `>`, `<`, …
- The time resolution can be set with by the function `sc_set_time_resolution(`*`double`*`, `*`sc_time_unit`*`)` (standard 1 PS)
- Special constant `SC_ZERO_TIME` (= `sc_time(0,SC_SEC)`)

```
sc_time name(1.5, SC_NS);
sc_time name2(name);
...
sc_start();
sc_start(name);
sc_start(sc_time(100,SC_US));
sc_stop();
...
sc_time name3 = sc_time_stamp();
...
```

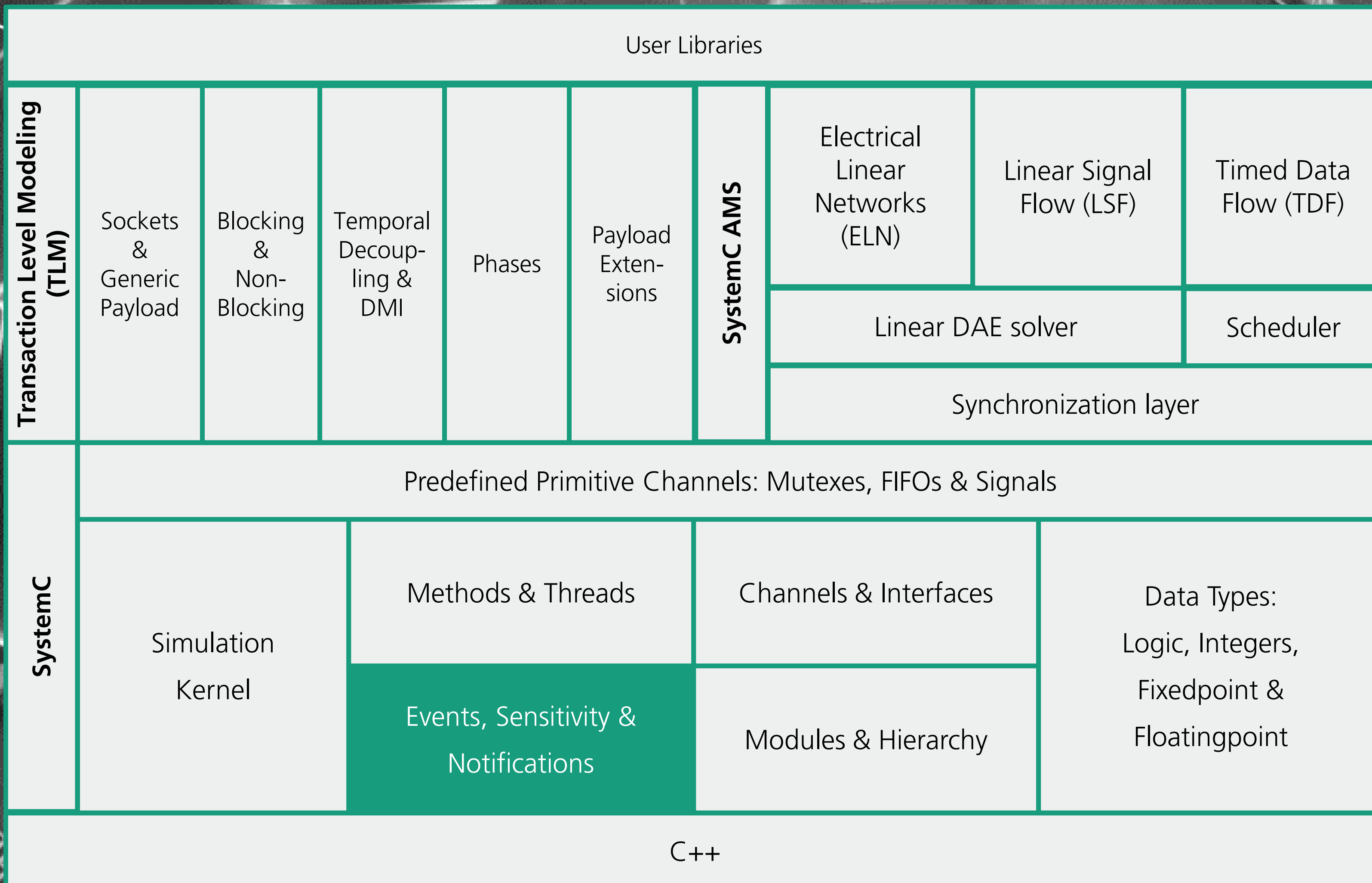
Simulation can run until there are no events, to a limited time, or unitl a call of `sc_stop()` in a process

The function `sc_time_stamp()` returns the current simulation time

enum	Units	Magnitude
SC_FS	Femtoseconds	10^{-15}
SC_PS	**Picoseconds**	$\mathbf{10^{-12}}$
SC_NS	Nanoseconds	10^{-9}
SC_US	Microseconds	10^{-6}
SC_MS	Milliseconds	10^{-3}
SC_SEC	Seconds	10^{0}

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++

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

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 Events: Sensitivity

- Static Sensitivity (RTL Style):
  - Is Specified in the constructor of the model (elaboration) for both, `SC_METHOD`s and `SC_THREAD`s
  - `sensitive << mySignal << myClock.pos() << myAwesomeEvent;`
  - Static sensitivity cannot be changed!

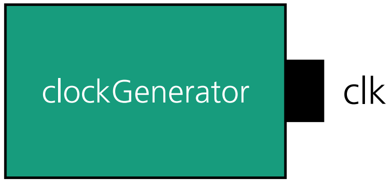
- Dynamic Sensitivity (TLM Style):
  - Dynamic Sensitivity lets a simulation process change its sensitivity on the fly by calling different functions within the process.
    - `SC_THREAD` uses `wait(myAwesomeEvent);`
    - `SC_METHOD` uses `next_trigger(myAwesomeEvent);`
  - The static sensitivity is <u>overwritten temporarily</u>.

SystemC’s Wait Statement

```
SC_MODULE(clockGenerator) {
    public:
    sc_out<bool> clk;
    bool value;
    sc_time period;

    SC_HAS_PROCESS(clockGenerator);
    clockGenerator(const sc_module_name &name, sc_time period) :
      sc_module(name), period(period), value(true)
    {
        SC_THREAD(generation);
    }
    void generation() {
        while(true) {
            value = !value;
            clk.write(value);
            wait(period/2);
        }
    }
};
```

```
wait();
wait(3);
wait(myEvent);
wait(sc_time(10,SC_NS));
wait(10,SC_NS);
wait(SC_ZERO_TIME);
```

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



- The wait function provides a syntax to allow to model delays within `SC_THREAD` processes.
- When a wait is invoked, the `SC_THREAD` process is suspended
- Waiting for integer e.g. 3 will wait 3 times
- Waiting for `SC_ZERO_TIME` will wait for one δ Cycle

Fraunhofer IESE

SystemC Events: Sensitivity

Static Sensitivity

THREAD

```
SC_MODULE (Module)
{
  sc_in<int> a;

  void process() {
    while(true) {
      wait();
      // do something
    }
  }

  SC_CTOR (adder)
  {
    SC_THREAD (process);
    sensitive << a;
  }
};
```

METHOD

```
SC_MODULE (Module)
{
  sc_in<int> a;

  void process()
  {
    // do something
  }

  SC_CTOR (adder)
  {
    SC_METHOD (process);
    sensitive << a;
  }
};
```

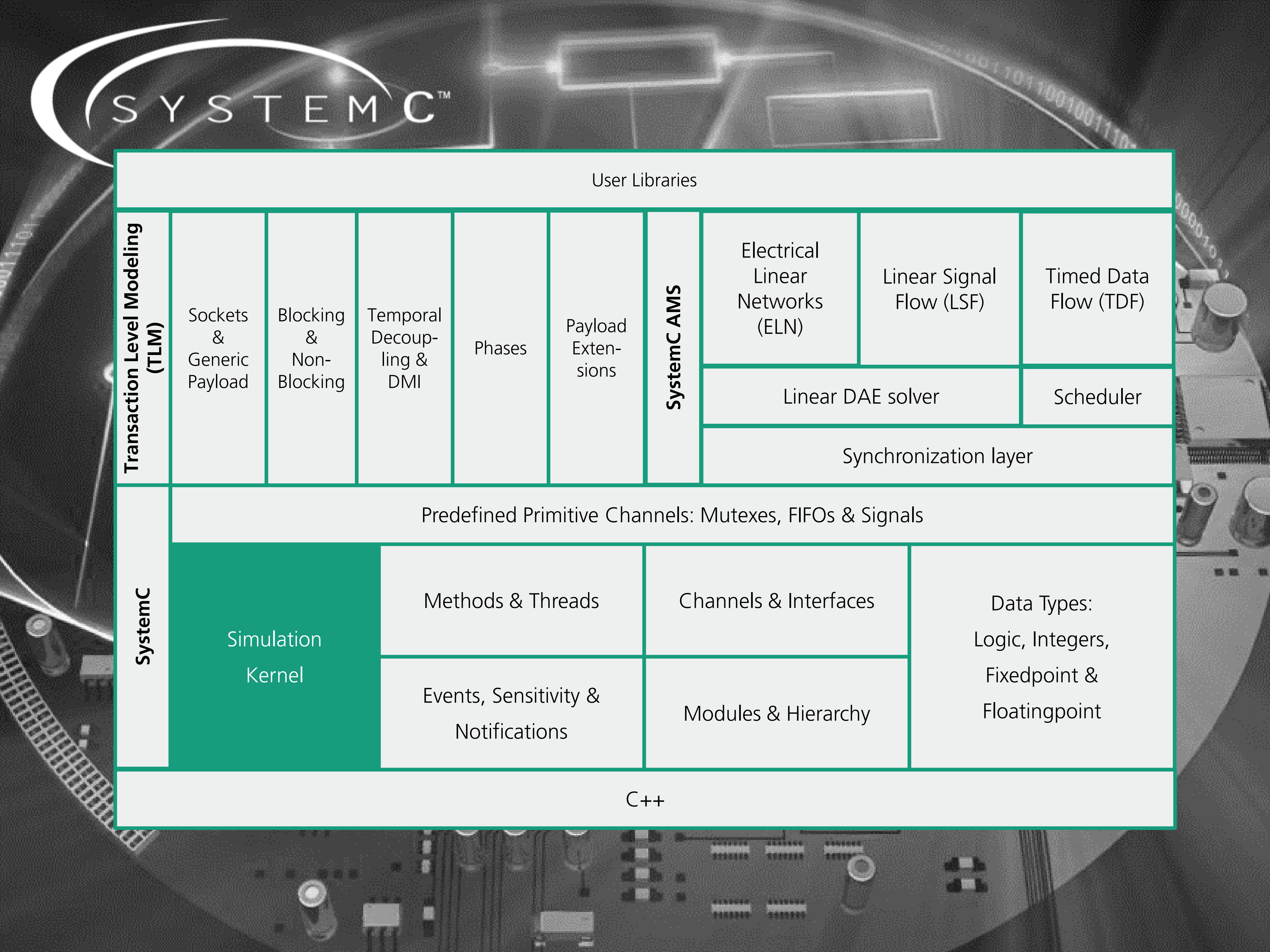
Dynamic Sensitivity

THREAD

```
SC_MODULE (Module)
{
  sc_in<int> a;

  void process() {
    while(true) {
      wait(
        a.valueChangedEvent
      );
      // do something
    }
  }

  SC_CTOR (adder)
  {
    SC_THREAD (process);
  }
};
```

METHOD

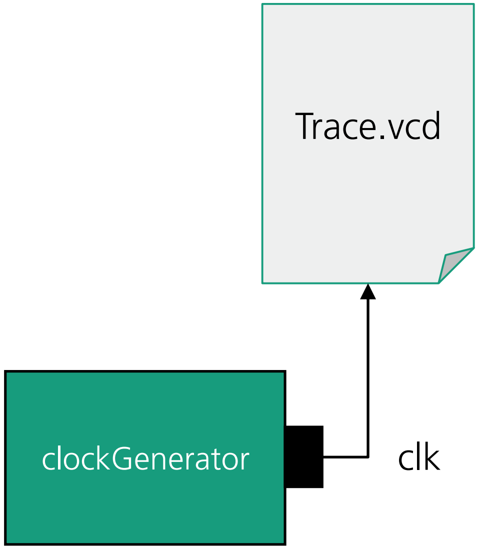
```
SC_MODULE (Module)
{
  sc_in<int> a;

  void process(
  {
    // do something
    next_trigger(
      a.valueChangedEvent
    );
  }

  SC_CTOR (adder)
  {
    SC_METHOD (process);
  }
};
```


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++

SystemC Waveform Tracing

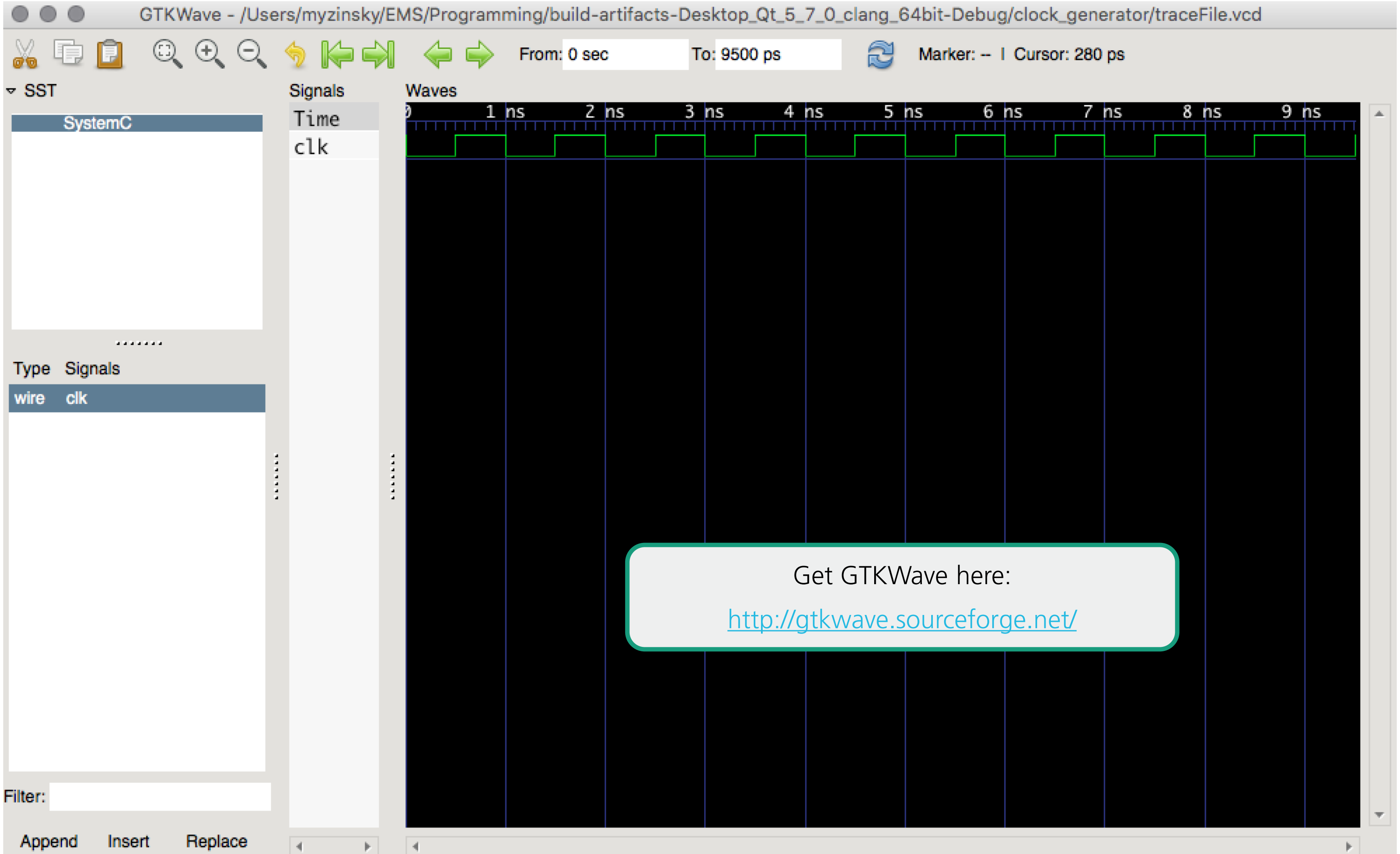
```
int sc_main ()
{
    clockGenerator g("clock_1GHz", sc_time(1,SC_NS));
    sc_signal<bool> clk;

    // Bind Signals
    g.clk.bind(clk);

    // Setup Waveform Tracing:
    sc_trace_file *wf = sc_create_vcd_trace_file("trace");
    sc_trace(wf, clk, "clk");

    // Start Simulation
    sc_start(10, SC_NS);

    // Close Trace File:
    sc_close_vcd_trace_file(wf);

    return 0;
}
```

- Like VHDL or Verilog, SystemC allows the non-intrusive recording of siganls into a waveform vcd file
- cout is printed in every delta cycle -> confusing



Fraunhofer IESE

SystemC Waveform Tracing



Get GTKWave here:

http://gtkwave.sourceforge.net/

SystemC's sc_clock

From SystemC Specification:

```
sc_clock(const char* name_,
        const sc_time& period_,
        double         duty_cycle_ = 0.5,
        const sc_time& start_time_ = SC_ZERO_TIME,
        bool           posedge_first_ = true );

sc_clock(const char* name_,
        double         period_v_,
        sc_time_unit   period_tu_,
        double         duty_cycle_ = 0.5 );

sc_clock(const char* name_,
        double         period_v_,
        sc_time_unit   period_tu_,
        double         duty_cycle_,
        double         start_time_v_,
        sc_time_unit   start_time_tu_,
        bool           posedge_first_ = true );
```

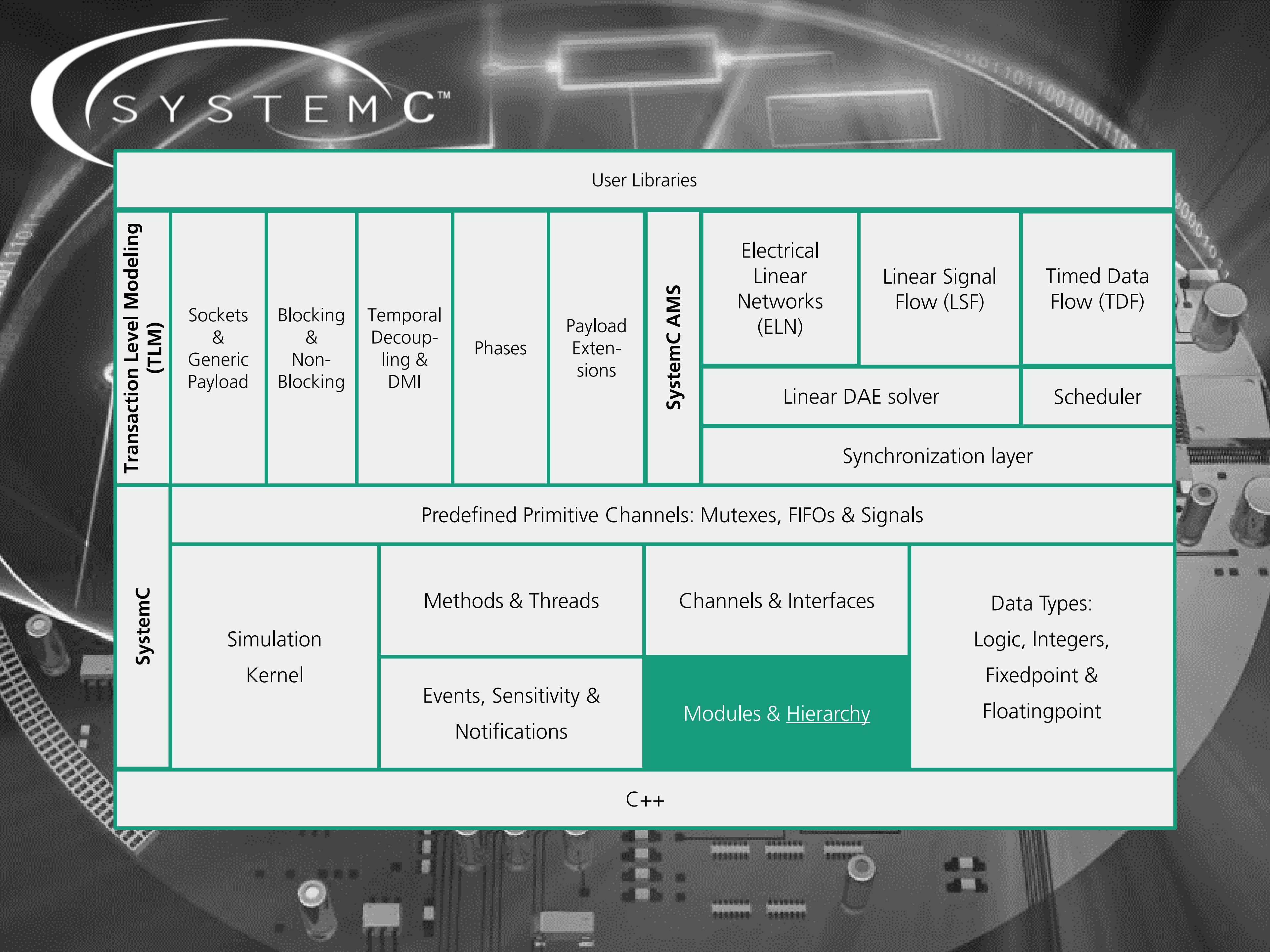
- For easy creation of clock generators
- Example:

```
sc_clock clock("Clk", 10, SC_NS, 0.5, 10, SC_NS);
sc_clock clock("Clk2", sc_time(10, SC_NS));
sc_clock clock("Clk3", 10, SC_NS, 0.5);
```

- Processes can be sensitive to clocks:

```
SC_METHOD(monitor);
sensitive << clk.pos();
```

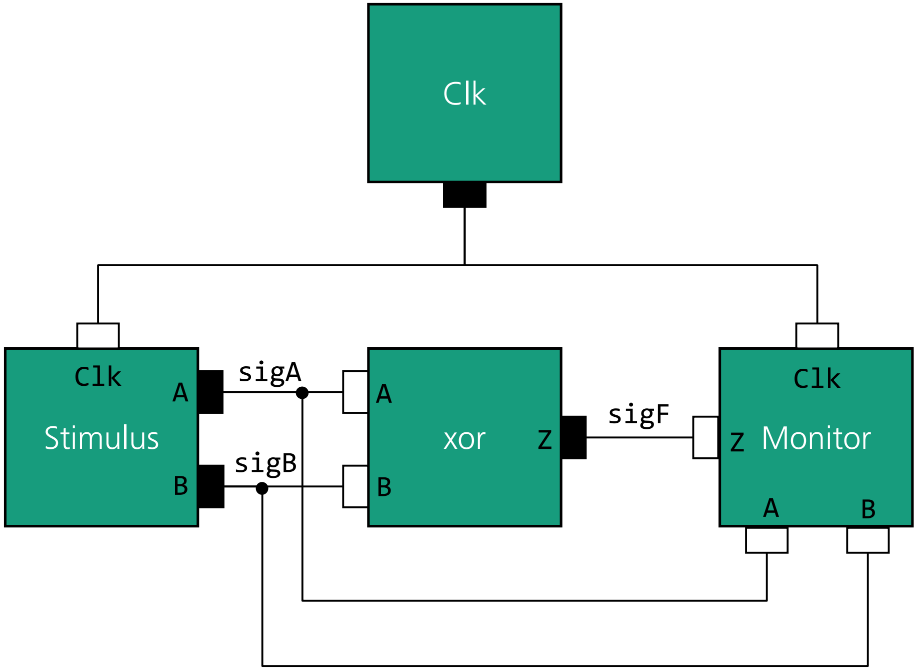

Fraunhofer IESE


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++

Connecting Modules (Binding)

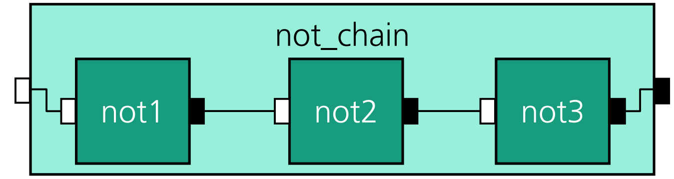
```
int sc_main(int argc, char* argv[]) {
    sc_signal<bool> sigA, sigB, sigF;

    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;
}
```

- Connecting `SC_MODULE`s in `sc_main` or in a toplevel module
- Binding of components with signals
- Keyword bind can be used or not



Fraunhofer IESE

Connecting Modules in Modules (Hierarchical Binding)




```
SC_MODULE(NOT)
{
  public:
  sc_in<bool> in;
  sc_out<bool> out;

  SC_CTOR(NOT) : in("in"), out("out")
  {
    SC_METHOD(process);
  }

  void process()
  {
    out.write(!in.read());
  }
};
```

```
SC_MODULE(not_chain) {
  sc_in<bool> A;
  sc_out<bool> Z;
  NOT not1, not2, not3;
  sc_signal<bool> h1,h2;

  SC_CTOR(not_chain):
  not1("not1"), not2("not2"),
  not3("not3"), A("A"), Z("Z"),
  h1("h1"), h2("h2")
  {
        not1.in.bind(A);
        not1.out.bind(h1);
        not2.in(h1);
        not2.out(h2);
        not3.in(h2);
        not3.out(Z);
    }
};
```

```
int sc_main ()
{
  sc_signal<bool> foo;
  sc_signal<bool> bar;

  not_chain c("not_chain");

  foo.write(0);
  c.A.bind(foo);
  c.Z(bar);

  sc_start();

  cout << bar.read(); //1
}
```

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

Fraunhofer IESE

Next Topics

- SystemC Data Types
- More on Modules and Hierarchy
- Ports (Exports, Multiports), Interfaces and Channels
- Event Queues, Event Finders
- Differences to VHDL
- Dynamic Processes
- Primitive Channels (FIFOs, Mutex …)
- Report Handling
- Callbacks (Elaboration…)
- Synthesis Subset / HLS

…

- Transaction Level Modelling (TLM)