SystemC and Virtual Prototyping

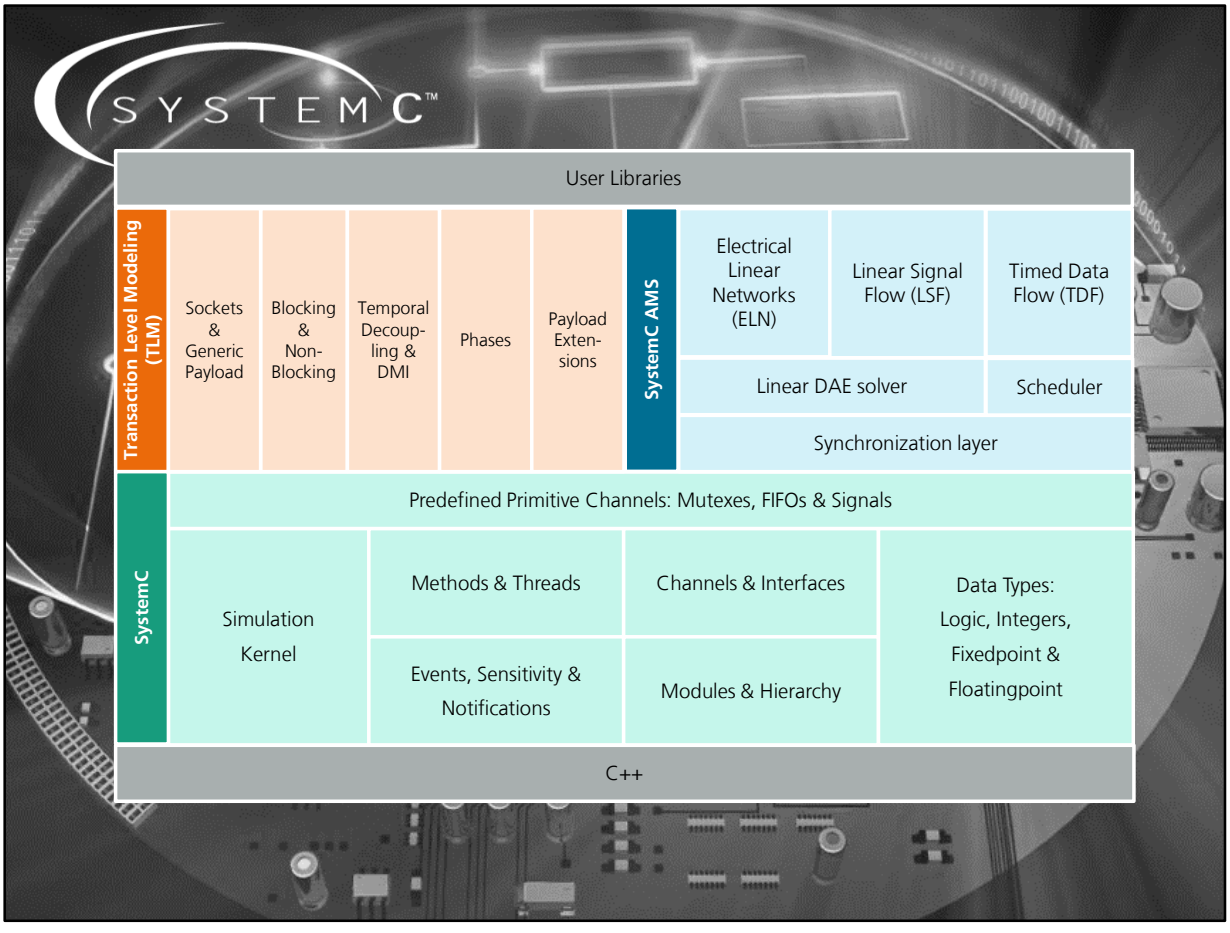
Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

Your notes:

About: Matthias Jung

- Studium Elektro- und Informationstechnik im Bereich der eingebetteten Systeme und Computerarchitektur
- Promotion über DRAM-Speicher, insbesondere mit dem Fokus auf schnelle und genaue Simulation mit SystemC
- 10 Jahre praktische Erfahrung im Bereich Virtual Prototyping sowohl in Forschungs- als auch in Industrieprojekten
- Lehrauftrag für die Vorlesung *SystemC and Virtual Prototyping* an der TU Kaiserslautern

Fraunhofer IESE

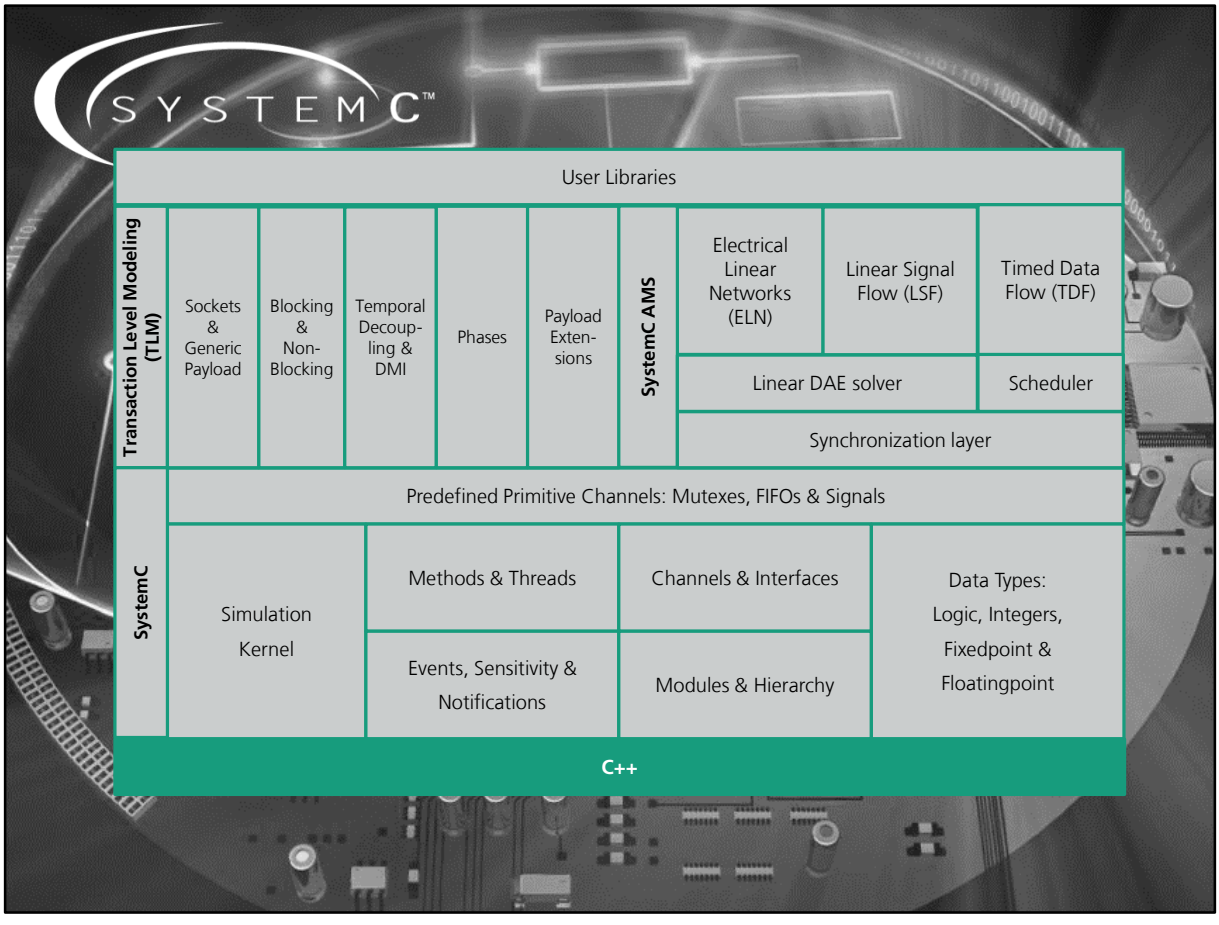

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:

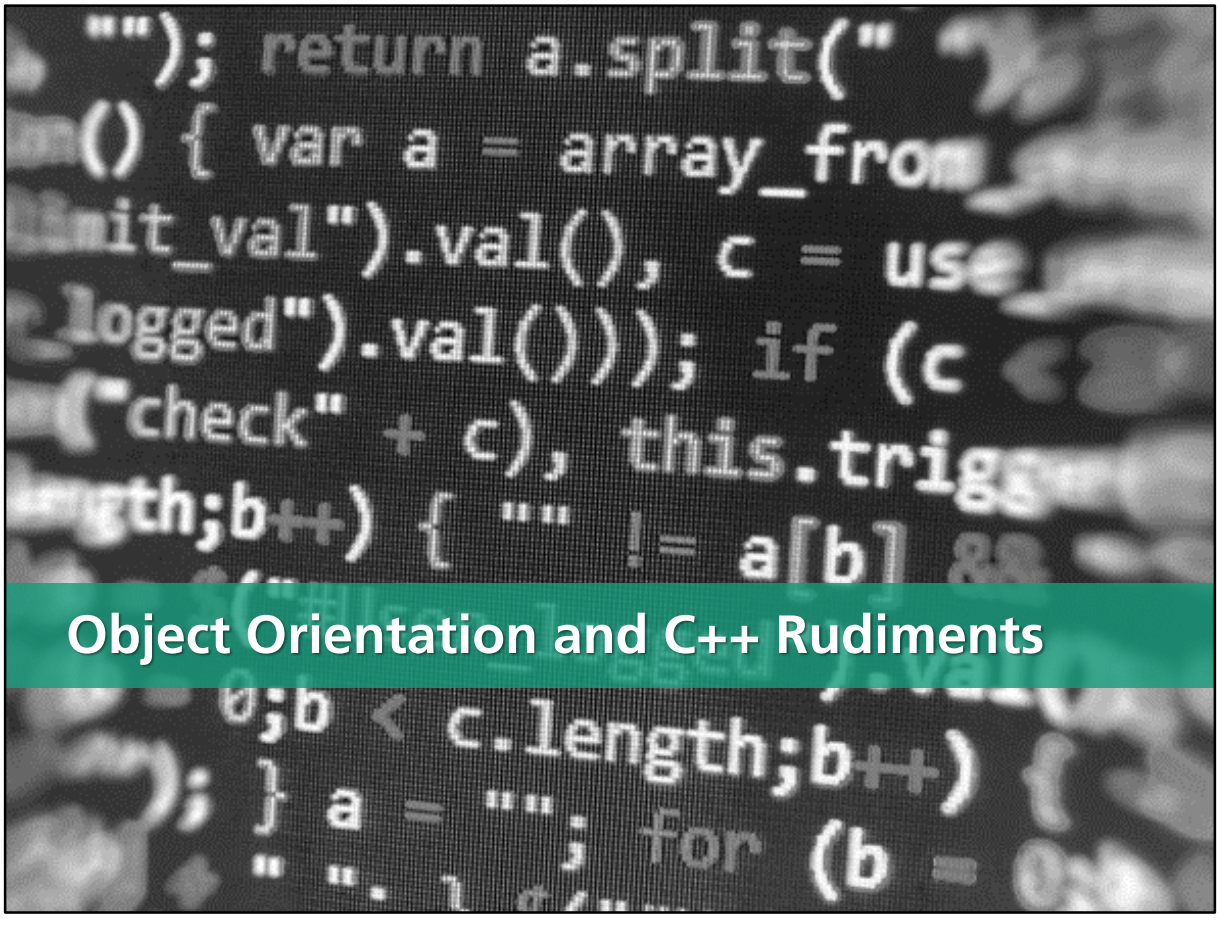
Time Planning

	Monday	Tuesday	Wednesday	Thursday	Friday
08:30	C++ and Introduciton to VP	SystemC Basics	SystemC Advanced	TLM Basics	TLM Advanced
12:00	Lunch	Lunch	Lunch	Lunch	Lunch
13:00	Exercise 0: Setup Artifacts	Exercise 1: Combinatorics XOR	Exercise 2: State Machine	Exercise 3: TLM LT and Routing	Exercise 4: TLM AT

Fraunhofer IESE

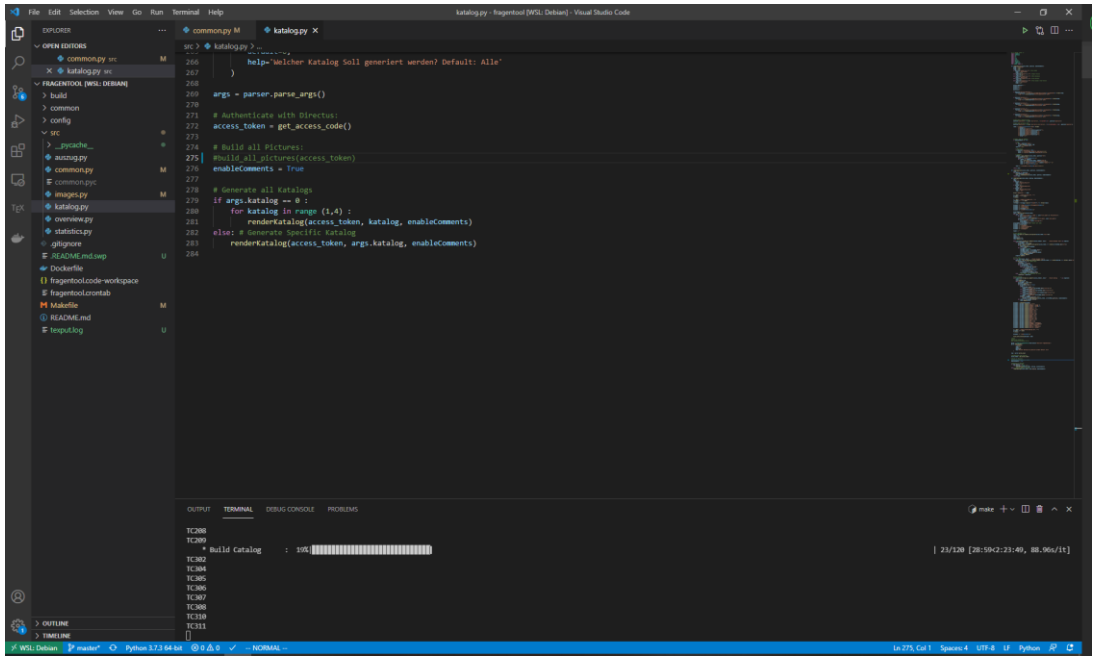

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:



Your notes:

IDE: VS Code



Fraunhofer IESE

Your notes:

C++ (ISO/IEC 14882:2014)

- General-purpose programming language
- Invented by Bjarne Stroustrup as "*C with Classes*"
- ++ means incremental to C
- Object Oriented
- C++ is standardized by an ISO working group

```
#include <iostream>

int main()
{
        std::cout << "Hello, world!" << std::endl;
        return 0;
}
```


Fraunhofer IESE

Your notes:

Simple & Artificial C++ Program

```cpp
#include <iostream>

void function(int i, int j)
{
    std::cout << i << " " << j << std::endl;
}


int main()
{
    // This is a comment
    int a = 5;
    int b = 6;

    /* This is another way to comment
    even over several lines */

    if(a == 7) {
        b = 10;
    } else if (a > 2 && a < 7) {
        b = 0;
    }

    for(int i = 0; i < b; i++) {
        a = a * i;
    }

    function(a, b);
}
```


Fraunhofer IESE

Your notes:

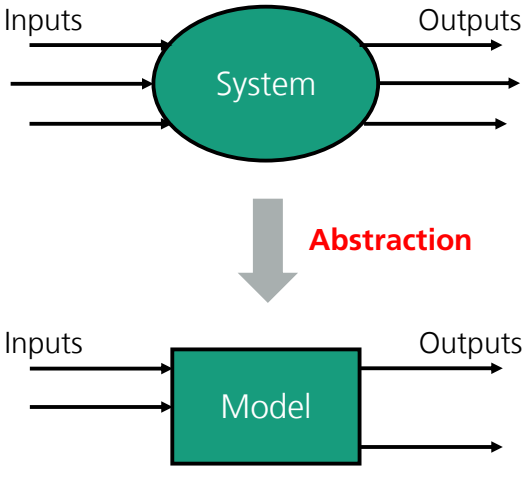
System and Model

- 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



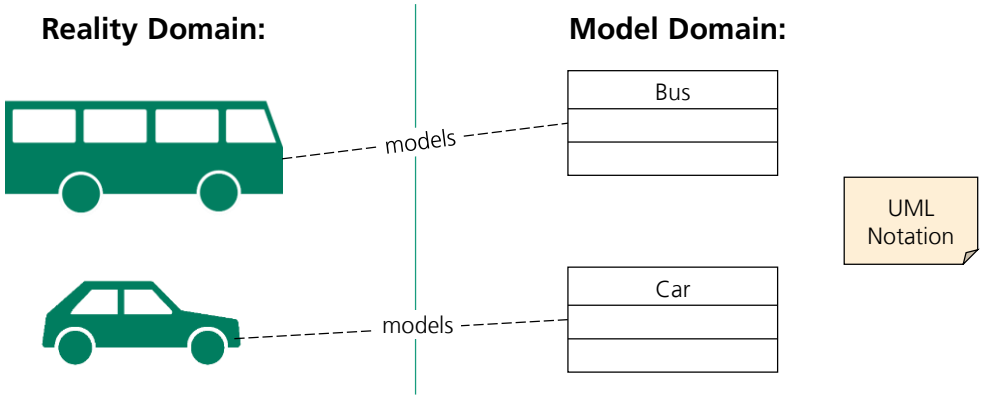
Fraunhofer IESE

Your notes:

Object Orientation (OO)

- Object: *Thing, Item, Article, Entity, Gadget, Gizmo, Widget, …*
- Orientation: *Direction, Coordination, Alignment, Configuration, …*

Object Orientation is the alignment on the things of reality!



11

Fraunhofer IESE

Object Orientation is the alignment on the things of reality! For instance, there exist vehicles in reality of different classes, for example Busses or Cars, which we can bring to the so called model domain.

Object-Oriented Programming (OOP) refers to a programming methodology based on objects, instead of just functions and procedures. These objects are organized into classes, which allow individual objects to be group together. Most modern programming languages including Java, C++, PHP and even SystemC are object-oriented languages, and many older programming languages now have object-oriented versions.

Your notes:

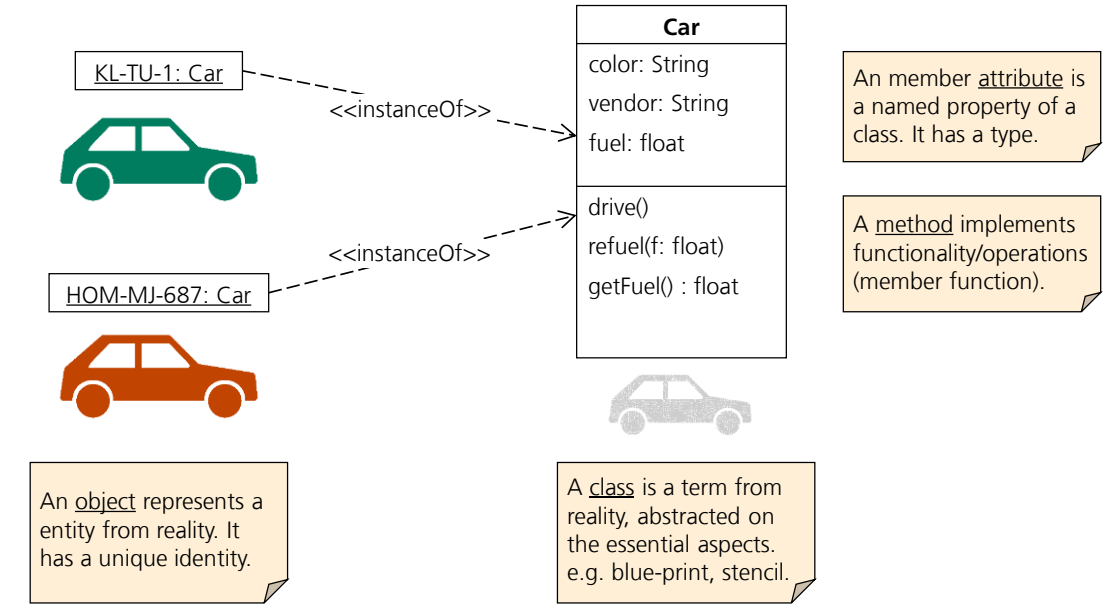
Object Orientation (OO)

- Abstraction is the key for OO! Abstraction trough:
  - Objects
  - Classes
  - Attributes
  - Methods
  - Encapsulation / Information Hiding
  - Inheritance
  - Static Members
  - Polymorphism
  - Templates

Fraunhofer IESE

Your notes:

Object, Classes, Attributes, Methods



13

Fraunhofer IESE

Abstraction: Things from the reality and our thinking are reduced to objects with essential properties. These objects are named, have a unique identity and can interact with each other. In order to avoid to describe all objects individually, we use classification. A class is a term from reality, abstracted on the essential aspects and can be seen as a mask, stencil (I avoid the word template because it is used in another context later). An object represents a entity from reality which can be classified into a class.

For example, two cars were reduced only on their number plate. The object HOM-MJ-687, is an instance of the class car, which has different properties. These properties are called member attributes, for example color or the amount of fuel. Furthermore, a class can provide functions that either return information about the attributes or modify the attributes. These member functions are usually called method

Your notes:

Object, Classes, Attributes, Methods in C++

```
#include <string>
#include <iostream>

class car
{
        // Member Variables:
        public:
        std::string color;
        std::string vendor;
        float fuel;

        // Member Functions (Methods):
        void drive();
        void refuel(float f);
        float getFuel()
        {
                return fuel;
        }

        // Constructor:
        car(std::string c, std::string v) : color(c), fuel(0)
        {
                vendor = v;
        }

        // Destructor:
        ~car(){…}
};
```

Class definition

Constructor/Destructor called when object is created / destroyed.

```
void car::drive()
{
        if(fuel > 10)
        {
                fuel = fuel -10;
        }
}

void car::refuel(float f)
{
        fuel = fuel + f;
}

int main()
{
        car kl_tu_1("green","VW");
        car * hom_mj_687 = new car("red","toyota");

        kl_tu_1.refuel(100);
        hom_mj_687->refuel(100);
        hom_mj_687->color = "green";
        std::cout << kl_tu_1.getFuel() << std::endl;

        delete hom_mj_687;
}
```

Methods can be defined within the class definition or outside

Pointers (->)

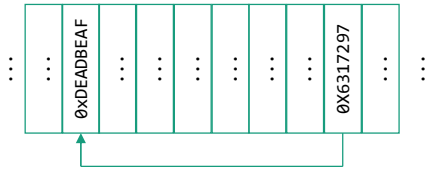
Object of class car are created by passing constructor arguments!

14

Fraunhofer IESE

Your notes:

Pointers, new and delete




```
int main()
{
    int var = 20;
    int *p;
    p = &var;
    std::cout << p << " " << *p << std::endl;

    car kl_tu_1("green","VW");
    car * hom_mj_687 = new car("red","toyota");

    delete hom_mj_687;

    unsigned int n;
    std::cin >> n;
    car * cars = new car[n];
    // Do something with the cars ...
    delete[] cars;
}
```

&: Referencing
*: Dereferencing

- A pointer is a variable whose value is the address of another variable
- Why the concept of **new**?
- Dynamic memory allocation instead of static memory allocation. Why?
- E.g. user can input size over command line etc. …
- Memory allocated dynamically is only needed during specific periods of time within a program. Once it is no longer needed it should be freed (**delete**) such that memory becomes available again.
- If you don't delete → memory leak

Fraunhofer IESE

Your notes:

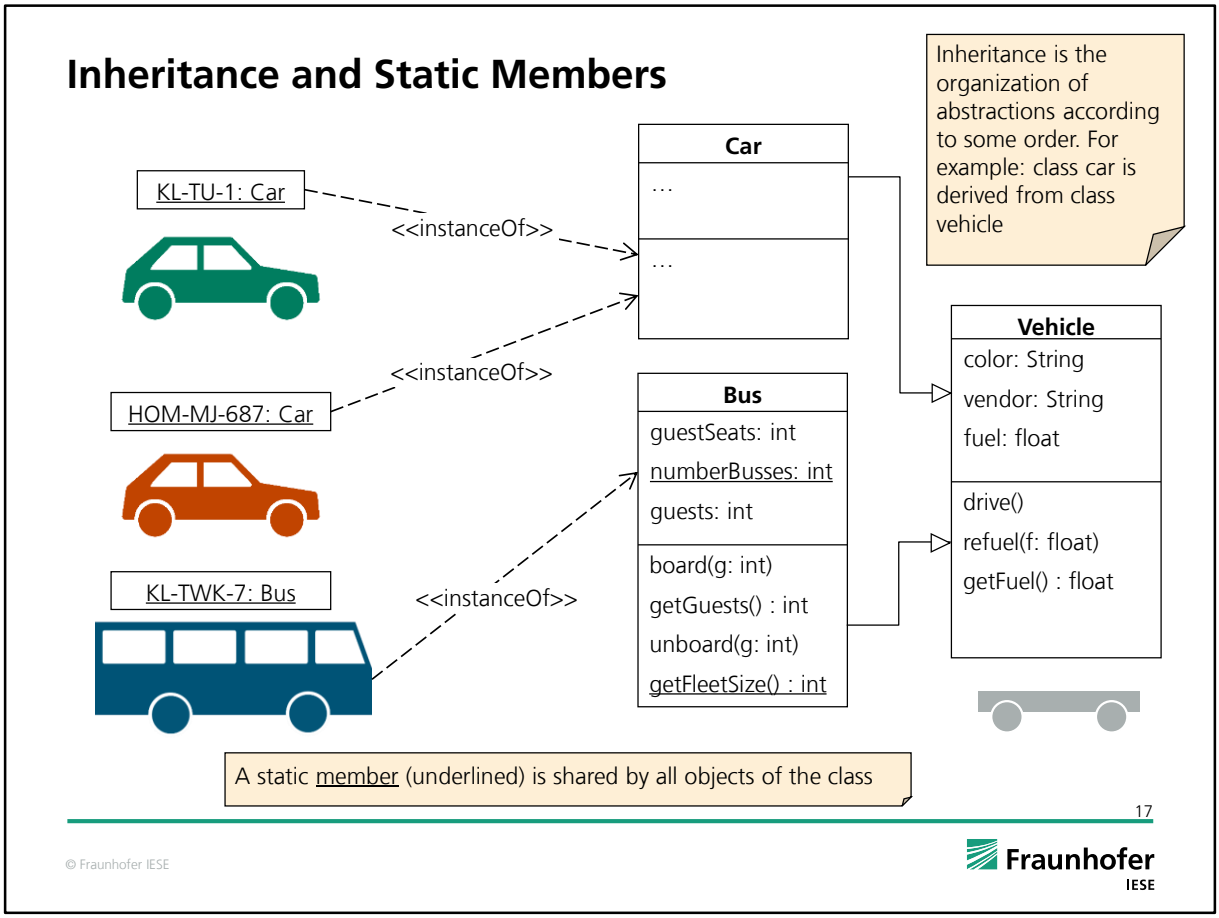
Call by Reference and Call by Value

```
#include<iostream>

void callByValue(int x) {
        x += 10;
}

void callByReference(int *x) {
        *x += 10;
}

void callByReference2(int &x) {
        x += 10;
}

int main() {
        int a=10;
        std::cout << "a = " << a << std::endl;
        callByValue(a);
        std::cout << "a = " << a << std::endl;
        callByReference(&a);
        std::cout << "a = " << a << std::endl;
        callByReference2(a);
        std::cout << "a = " << a << std::endl;
        return 0;
}
```

- ***Call by Value*:** If data is passed by value, the data is copied from the variable used in for example `main()` to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.
- ***Call by Reference*:** If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in `main()`.
- Heavily used in SystemC Transaction Level Modelling (TLM)

Fraunhofer IESE

Your notes:



One important characteristic of object-oriented languages is inheritance. Inheritance refers to the capability of defining a new class that inherits methods and attributes from a parent class, i.e. the code for the attributes and methods has not to be rewritten. New attributes and methods can be added to the new class.

In UML inheritance is shown with arrows pointing from the child to the parent with a unfilled arrow head.

Inheritance can help to organize abstractions. For example, there could be a more generic base class called vehicle, from which the classes car and bus can inherit all common properties.

Classes can have static member variables and functions. For example a static member variable exists only once and is shared globally for all members of this class. In UML static members are underlined.

Your notes:

Inheritance in C++

```cpp
#include <string>
#include <iostream>

class vehicle
{
    // Member Variables:
    public:
    std::string color;
    std::string vendor;
    float fuel;

    // Member Functions (Methods):
    void drive(){…};
    void refuel(float f){…};
    float getFuel(){…};

    // Constructor:
    vehicle(std::string c, std::string v) :
        color(c), fuel(0)
    {
        vendor = v;
    }
};
```

```cpp
class bus : public vehicle {
    // Additional Member Variables:
    public:
    int guestSeats;
    static int numberBusses;
    int guests;

    // Additional Member Functions:
    void board(int g){…};
    int getGuests(){…};
    void unboard(int g){…};
    static int getFleetSize(){ return numberBusses; };

    // Constructor
    bus(std::string c, std::string v, int g) : vehicle(c,v), guestSeats(g) {
        numberBusses++; // increase number of busses when a new is created
    }
};

// Initialize static member of class bus:
int bus::numberBusses = 0;

int main() {
    bus kl_twk_1("blue","mercedes",56);
    bus kl_twk_2("red", "mercedes",58);
    std::cout << bus::getFleetSize() << std::endl;
    std::cout << bus::numberBusses << std::endl;
    std::cout << kl_twk_2.getFleetSize() << std::endl;
}
```

Inheritance happens here!

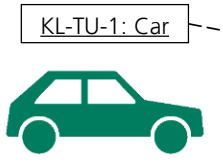
Static variable and method

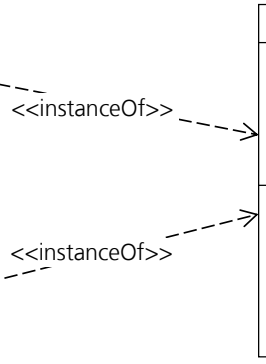
Output will be:
2
2
2

Fraunhofer IESE

Your notes:

Encapsulation / Visibility / Information Hiding





The *private* member are not accessible outside the class. They can be accessed only trough methods of the class. The *public* members are accessible outside the class. The *protected* members can be accessed by inherited classes.

Attribute	Inside Class	Inherited Class	Other Class / Outside
+ public	X	X	X
# protected	X	X	
– private	X		

We will ty to keep the visibility as minimal as possible!

19

Fraunhofer IESE

The idea of Encapsulation is the localization of features into a single blackbox abstraction that hides their implementation details behind a public interface. This concept is also often called as "Information Hiding". It is intended to keep the visibility as minimal as possible in order to concentrate only on the essential aspects!

Therefore, OO provides usually three different classifiers for visibility. The *private* (-) member are not accessible outside the class. They can be accessed only trough methods of the class (usually get and set methods are used for that). The *public* members (+) are accessible outside the class. The *protected* members (#) can be accessed by inherited classes.

Your notes:

Access Variables of Parent Class

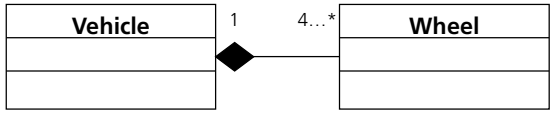
```
class Base {
public:
  Base(): a(0) {}
  virtual ~Base() {}
protected:
  int a;
};

class Child: public Base {
public:
  Child(): Base(), b(0) {}
  void foo();

private:
  int b;
};

void Child::foo() {
  b = Base::a; // Access variable 'a' from parent
}
```

- The *private* member are not accessible outside the class.
- They can be accessed only trough methods of the class.
- The *public* members are accessible outside the class.
- The *protected* members can be accessed by inherited classes.
- Parental member can be accessed with the parents class name in the front followed by `::`
- It is a good practice to make member variables always private and use public member functions to set and get their values

Using Classes as Members

```
#include <string>
#include <iostream>

class wheel
{
    public:
    wheel() {}
};

class vehicle
{
    private:
    wheel * wheels;

    public:
    vehicle(int numberOfWheels)
    {
        if(numberOfWheels >= 4) {
            wheels = new wheel[numberOfWheels];
        } else {
            wheels = NULL;
            std::cout << "Invalid Wheels Setup" << std::cout;
        }
    }

    ~vehicle()
    {
        delete [] wheels;
    }
};
```

Destructor is used for cleanup!




```
int main()
{
    vehicle v1(3);
    vehicle v2(4);
}
```

- Classes can be composed out of other classes
- This will be heavily used in SystemC based designs

Fraunhofer IESE

Overloading Methods

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      void area() {
         cout << "Base class, no area!" << endl;
      };
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      void area () {
         cout << "Rectangle class area :"
              << (width * height) << endl;
      }
};
```

```
class Triangle: public Shape {
   public:
      Triangle(int a = 0, int b = 0):Shape(a, b) { }

      void area () {
         cout << "Triangle class area : "
              << (width * height / 2) << endl;
      }
};

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

   shp.area();
   rec.area();
   tri.area();

   return 0;
}
```

Output:
Base class, no area!
Rectangle class area: 50
Triangle class area: 25

Fraunhofer IESE

Your notes:

Polymorphism – Motivation

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      void area() {
         cout << "Base class, no area!" << endl;
      };
};
```

```
[ ... ]

// 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;
}
```

& referencing: taking the address of an existing variable or object.

Output:
Base class, no area!
Base class, no area!

- During runtime we want be flexible and work with the base class!

Fraunhofer IESE

Your notes:

Polymorphism – Virtual Functions

```
#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() {
         cout << "Base class, no area!" << endl;
      };
};
```

```
[ ... ]

// 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

- Polymorphism gives us the ability to switch components without loss of functionality
- If child class does not implement virtual function the base method is called

24

Fraunhofer IESE

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. In C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Your notes:

Polymorphism – Pure Virtual (Abstract Base Classes)

```
#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;
};
```

```
[ ... ]

// 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 pointers 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 → Exchangeability during Runtime

25

Fraunhofer IESE

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:

Operator Overloading

```
class Complex {
    private:
      double real;
      double imag;
    public:
       Complex(double r=0, double i=0): real(r), imag(i) { }

       // Operator overloading
       Complex operator + (Complex c2) {
           Complex temp;
           temp.real = real + c2.real;
           temp.imag = imag + c2.imag;
           return temp;
       }

       void output() {
           if(imag < 0)
               cout << "Complex: " << real << imag << "i" << endl;
           else
               cout << "Complex: " << real << "+" << imag << "i" << endl;
       }
};
```

```
int main()
{
    Complex c1(1,-2), c2(1,1), result;

    result = c1 + c2;
    result.output();

    return 0;
}
```

Output:
Complex: 2-1i

- Overloading operators allows to use custom classes like normal datatypes
- Very useful for simple and structured writing of code. SystemC uses this feature extensively.

Fraunhofer IESE

Your notes:

Templates

```
#include<iostream>

template<typename TYPE>
class adder
{
    public:
    TYPE lastResult;
    TYPE add(TYPE a, TYPE b)
    {
        lastResult = a + b;
        return lastResult;
    }
};

int main()
{
    adder<int> a1;
    adder<float> a2;

    int res1 = a1.add(5, 4);
    float res2 = a2.add(5.5, 4.4);

    std::cout << "res1 = " << res1 << std::endl;
    std::cout << "res2 = " << res2 << std::endl;

    return 0;
}
```

Also keyword `class` will work instead of `typename`

- **Function templates** are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
- **Class templates** have members that use template parameters as types.

Fraunhofer IESE

Your notes:

Templates

```
#include<iostream>

template<int MOD=4>
class counter
{
    public:
    int cnt;

    counter() : cnt(0) {}

    void count()
    {
        cnt = (cnt+1) % MOD;
    }
};
```

The default value is optional

```
int main()
{
    counter<2> c1;
    counter<> c2;

    for (int i = 0; i < 6; i++) {
        std::cout << "c1=" << c1.cnt << std::endl;
        c1.count();
    }

    for (int i = 0; i < 6; i++) {
        std::cout << "c2=" << c2.cnt << std::endl;
        c2.count();
    }

    return 0;
}
```

Non-type parameters for templates:

Templates can also have regular typed parameters, similar to those found in functions.

Fraunhofer IESE

Your notes:

C++ Standard Template Library (STL)

- **Containers**
  Containers are used to manage collections of objects of a certain kind. There are several different types of containers like `deque`, `list`, `vector`, `map` etc.
- **Algorithms**
  Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
- **Iterators**
  Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

Your notes:

STL Containers

- Sequence containers:
  - `array` - Array
  - `vector` - Vector
  - `deque` - Double ended queue
  - `forward_list` - Forward list
  - `list` - List
- Container adaptors:
  - `stack` - LIFO stack
  - `queue` - FIFO queue
  - `priority_queue` - Priority queue
- Associative containers:
  - `set` - Set
  - `multiset` - Multiple-key set
  - `map` - Map
  - `multimap` Multiple-key map
- Unordered associative containers:
  - `unordered_set` - Unordered Set
  - `unordered_multiset` – U. Multiset
  - `unordered_map` - Unordered Map
  - `unordered_multimap` - Unordered Multimap

30

Fraunhofer IESE

Your notes:

Example STL vector

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

int main() {

   // create a vector to store int
   vector<int> vec;
   int i;

   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++)
   {
      vec.push_back(i);
   }
```

```
   // display extended size of vec
   cout << "extended vector size = "
        << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++)
   {
      cout << "value of vec [" << i << "] = "
           << vec[i] << endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end())
   {
      cout << "value of v = " << *v << endl;
      v++;
   }
   return 0;
```

http://www.cplusplus.com/reference/

Fraunhofer IESE

Your notes:

Algorithms

- adjacent_find
- all_of
- any_of
- binary_search
- copy
- copy_backward
- copy_if
- copy_n
- count
- count_if
- equal
- equal_range
- fill
- fill_n
- find
- find_end
- find_first_of
- find_if
- find_if_not
- for_each
- generate
- generate_n
- includes
- inplace_merge
- is_heap
- is_heap_until
- is_partitioned
- is_permutation
- is_sorted
- is_sorted_until
- iter_swap
- lexicographical_comp
- lower_bound
- make_heap
- max
- max_element
- merge
- min
- minmax
- minmax_element
- min_element
- mismatch
- move
- move_backward
- next_permutation
- none_of
- nth_element
- partial_sort
- partial_sort_copy
- partition
- partition_copy
- partition_point
- pop_heap
- prev_permutation
- push_heap
- random_shuffle
- remove
- remove_copy
- remove_copy_if
- remove_if
- replace
- replace_copy
- replace_copy_if
- replace_if
- reverse
- reverse_copy
- rotate
- rotate_copy
- search
- search_n
- set_difference
- set_intersection
- set_symmetric_diff
- set_union
- shuffle
- sort
- sort_heap
- stable_partition
- stable_sort
- swap
- swap_ranges
- transform
- unique
- unique_copy
- upper_bound

```
std::sort (myvector.begin(), myvector.end());
```


Fraunhofer IESE

Your notes:

C++ 11, 14 …

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {0, 1, 2, 3, 4, 5};

    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << ' ';
    }

    std::cout << std::endl;

    for (auto i : v)  {
        std::cout << i << ' ';
    }
    std::cout << std::endl;

    std::vector<std::vector<std::vector<int>>> foo;

    auto bar = foo;
}
```

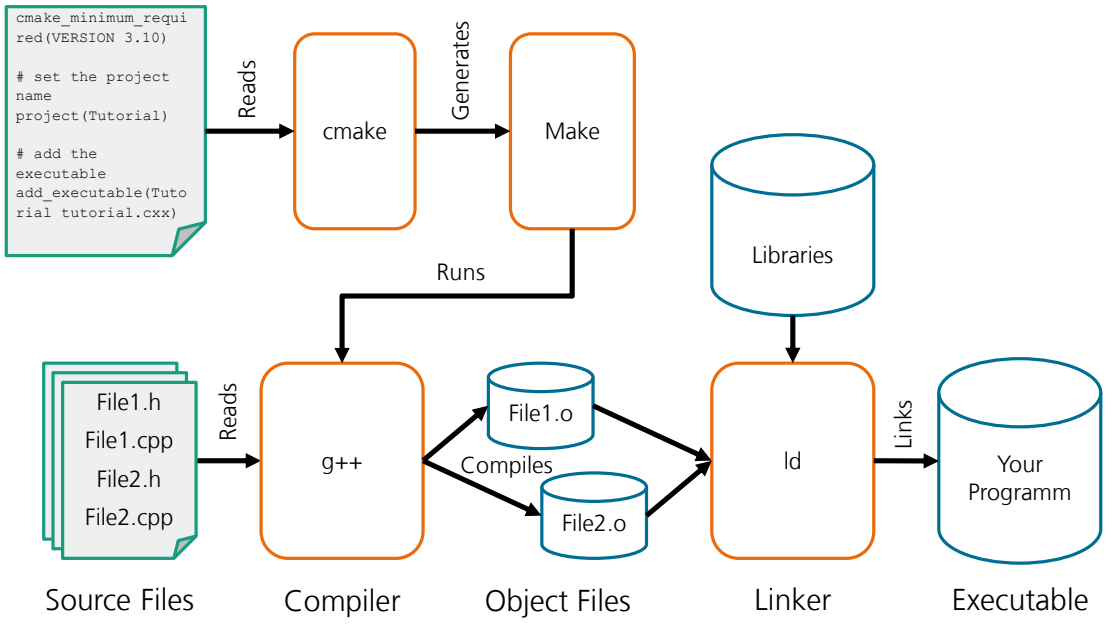
Whats new in C++ 11 …

- Lambda Expressions. …
- Automatic Type Deduction and decltype. …
- Uniform Initialization Syntax. …
- Deleted and Defaulted Functions. …
- nullptr. …
- Delegating Constructors. …
- Rvalue References. …
- C++11 Standard Library.

Fraunhofer IESE

Your notes:

Configure your project with CMake



Fraunhofer IESE

Your notes:

SystemC and Virtual Prototyping

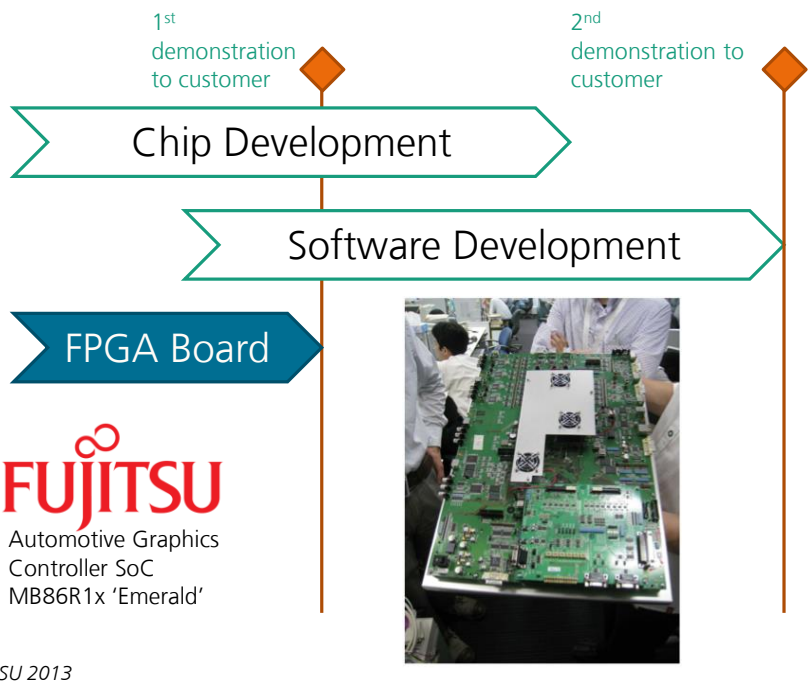
Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

Your notes:

Prototyping in Industry Nowadays

Your notes:

State of the Art – Prototyping Example



Source: FUJITSU 2013

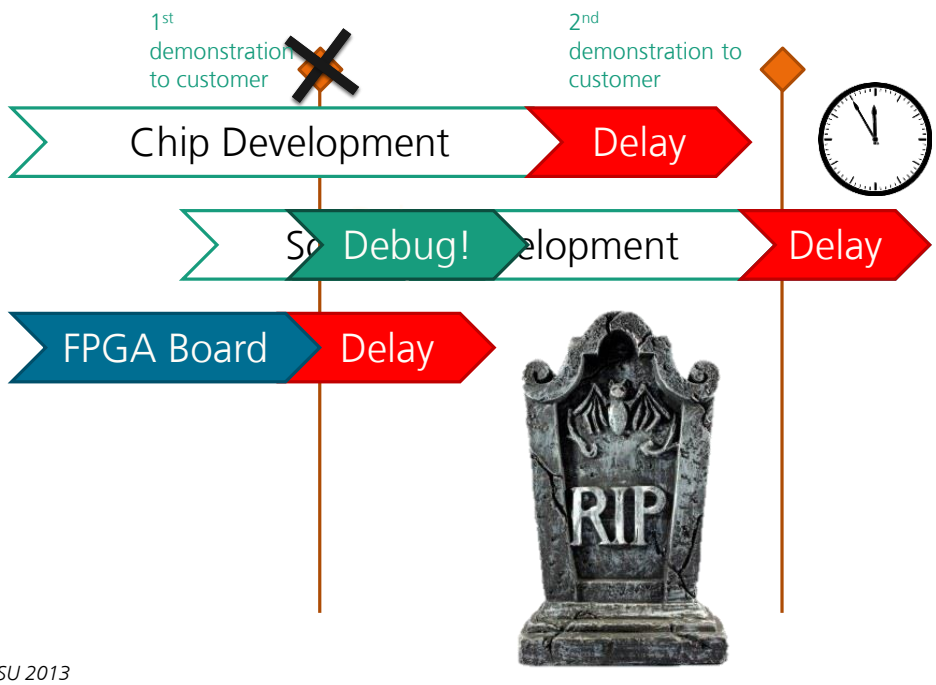
37

Fraunhofer IESE

Today's companies have to deal with complex hardware architectures like the before presented multi-core systems. Moreover, they have a constant pressure to deliver their products quickly because of many competitors on the market. The old-established design-flow procedures have a performance problem due to the high complexity of modern systems. New development tools and approaches for *Electronic System Level* (ESL) design are needed to fulfil these requirements. In the past the software was developed after the hardware as available. To overcome these gaps hardware-teams create rapid prototypes by means of e.g. a *Field Programmable Gate Array* (FPGA) to develop software in an environment similar to the target hardware architecture.

Your notes:

State of the Art – Prototyping Example - Reality



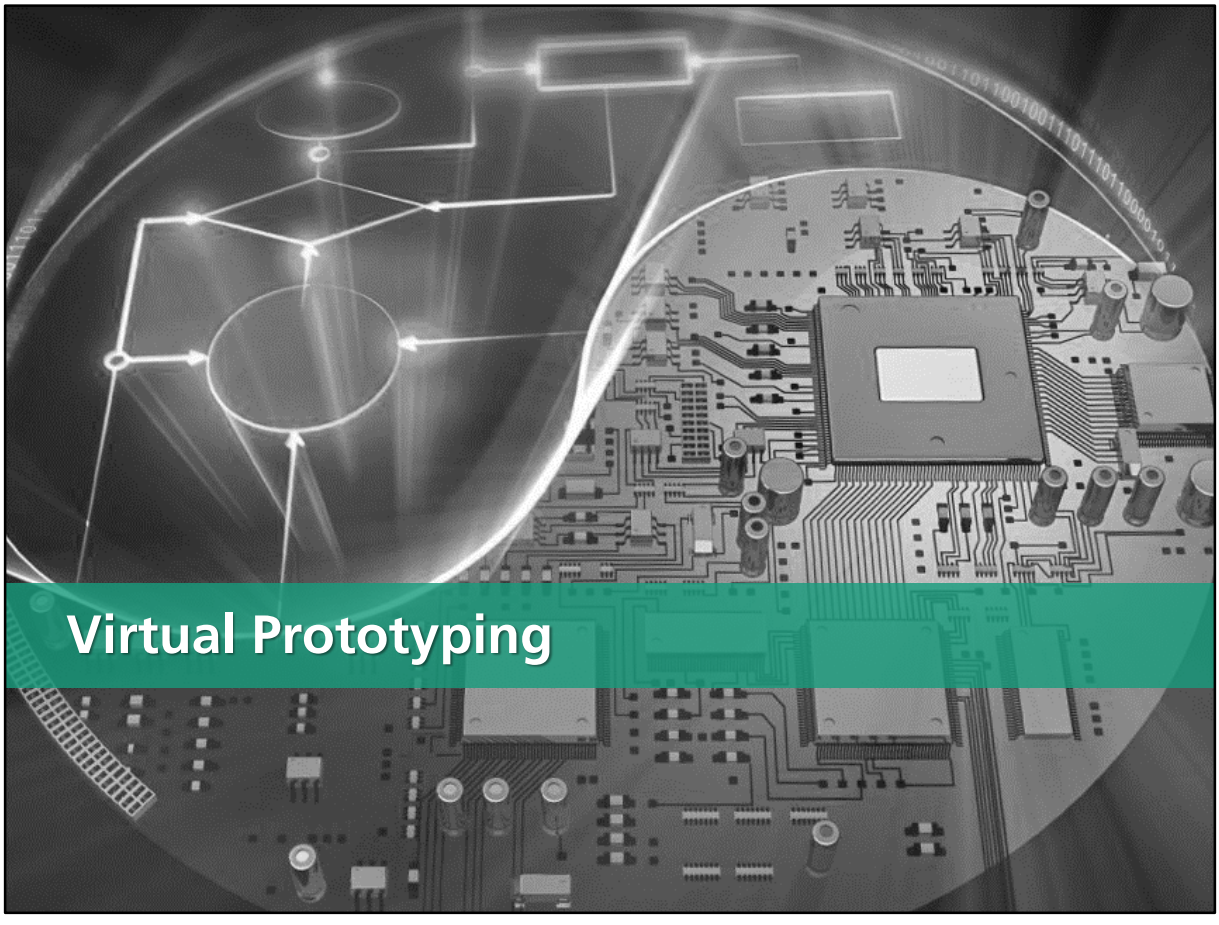
Source: FUJITSU 2013

38

Fraunhofer IESE

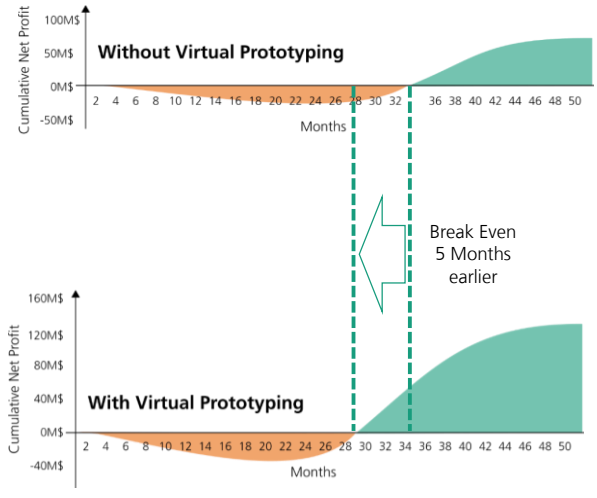
However, FPGA development can be also challenging. In this Example the development of the FPGA board was delayed, such that the final deadline was missed.

Your notes:



Your notes:

Shift Left with Virtual Prototyping



- Hedge decisions early
- Bugs are found and fixed early (Typically 56% of bugs emerge due to wrong requirements)
- Saving Time to Market (TTM) and resources
- Allowing good test coverage
- Better team work with HW engineers, SW developers and testers – this model minimizes frictional differences between them
- Delivery of software is accelerated
- Cost effectiveness

40

Source: Better Software. Faster! Tom De Schutter et al. March 2014, Synopsys Press

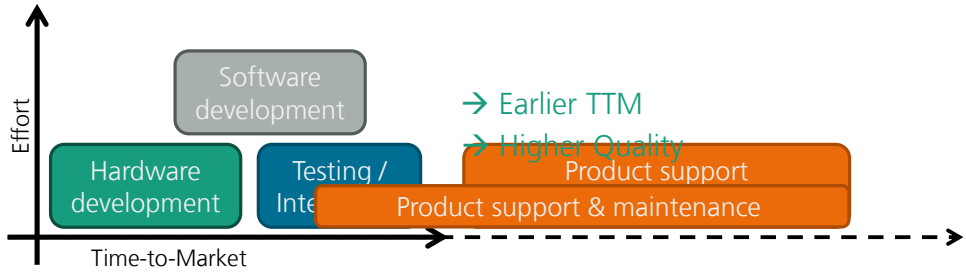
Fraunhofer IESE

To master this situation of complex hardware/software and the pressure with respect to *Time to Market* (TTM) a new idea has emerged: the idea of parallel development of hardware and software by means of *Virtual Prototypes* (VP) often denoted as *Shift Left.* Virtual prototypes are high-speed, fully functional software models of physical hardware systems, which are used for software development before the actual hardware is available.

Your Notes:

Virtual Prototypes

- High-speed functional software models of physical hardware
- Concurrent HW and SW development
- Design Space Exploration (DSE) for HW
- Visibility and controllability over the entire system
- Powerful debugging and analysis tools
- Reuse of components for future projects
- Teams can use it world wide
- Continuous Integration and Validation



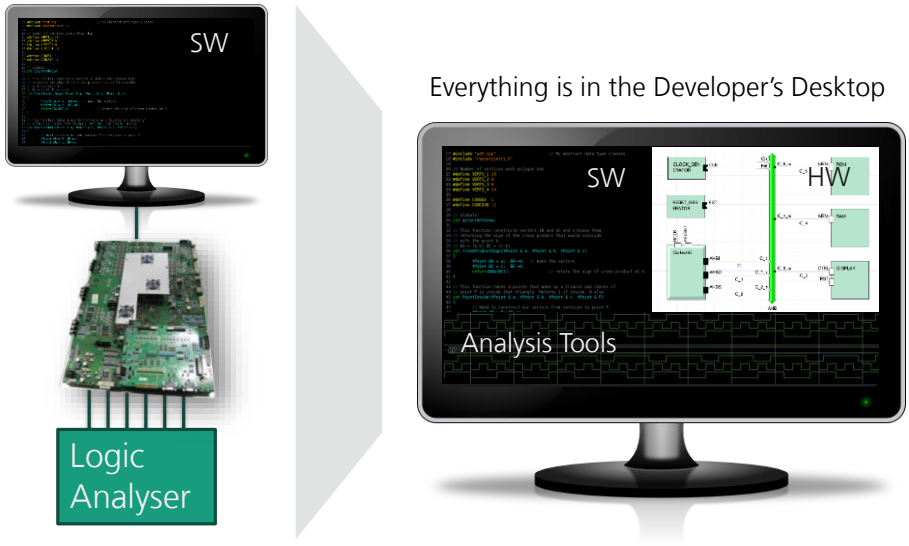
41

Fraunhofer IESE

To decrease the *Time-to-Market* (TTM), costs and efforts, it is necessary to develop software and hardware more concurrently and a support of the collaboration of the hardware and the software developer teams is mandatory. An effective approach for this issue is called *Virtual Prototyping* (VP). Virtual prototypes are high-speed, fully functional software models of physical hardware systems which can model a complete MPSoCs with reasonable simulation speed. It is easier to test the product because the virtual prototype provides visibility and controllability over the entire system. There are helpful and powerful debugging mechanisms for virtual prototypes which are almost unthinkable on a real hardware system. This leads to a higher quality of the product and a lower supporting effort. The slide shows the fusion of hardware development, software development and testing which leads to a decreased TTM, less effort, better quality, less costumer support, and finally to reduced costs. Case studies have shown that it is possible to deliver more competitive products up to 6 months faster.

Virtual Prototypes are well suited for early software development. However, they are also reasonable for hardware *Design Space Exploration* (DSE) because of easy modification and fast simulation speed. DSE is the investigation of different implementation variants regarding their optimal solution. For example: with a virtual prototype it can be easily explored how many cores are needed for a MPSoC to fulfill the requirements of the application.

Move to Virtuality?



42

Fraunhofer IESE

The problem of traditional hardware development is the slowness of the *Register Transfer Level* (RTL) simulation and the lack of configurability and visibility to analyze performance. In the traditional software development the visibility and controllability over the entire system is often missed as well. The programmer is limited to a JTAG or RS232 interface or has to use a logic analyzer for debugging. With virtual prototypes the hardware, software, toolchain and debugging tools are located in the developers desktop PC, as shown in the slide. The developer can easily observer all internal register, signals and pins.

Your notes:

Virtual Prototyping will Help!

Chip Development

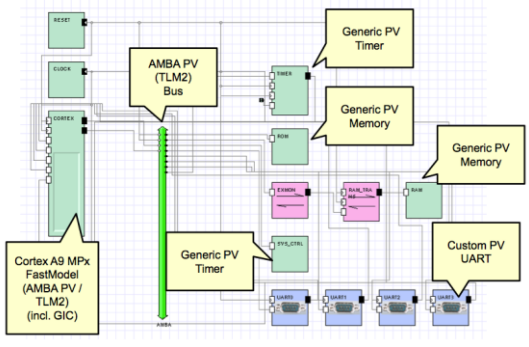
Software Development

Virtual Prototype

2 days after silicon 'Quake' was running

FUJITSU

Automotive Graphics Controller SoC MB86R1x 'Emerald'



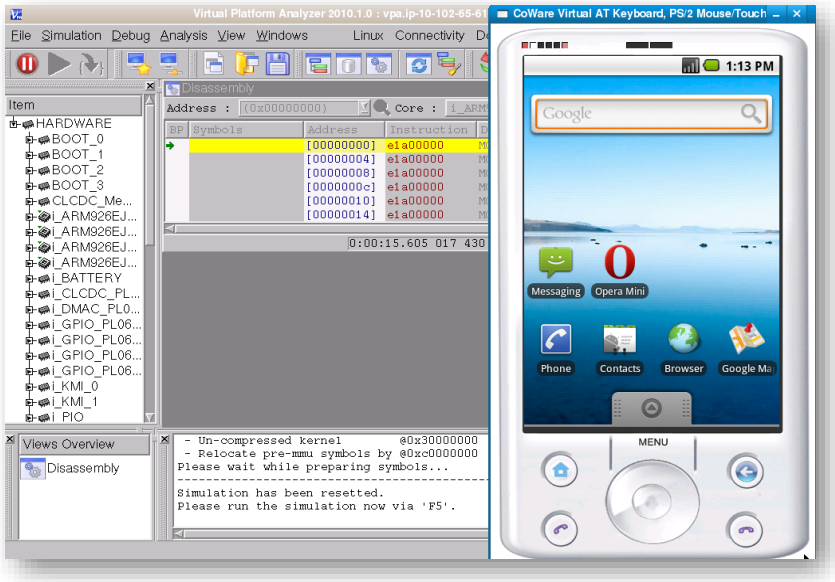
43

Fraunhofer IESE

Virtual Prototyping helped in our example to overcome the expensive development of prototyping boards. The basic platform components have been modeled with SystemC models and all the driver and base software could already be developed. It took only two days effort to port the software from the VP to the hardware.

Your notes:

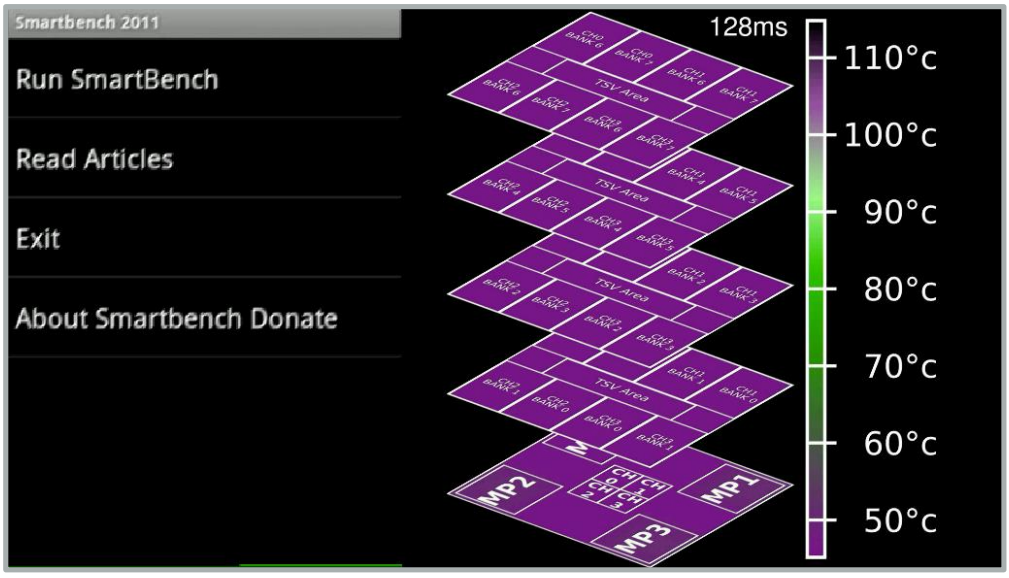
How to build a virtual Smartphone?



Fraunhofer IESE

Your notes:

Simulate Future Smartphone with 3D Integrated Circuits



45

Fraunhofer IESE

Your notes:

Accuracy vs. Speed Trade-Off

Fraunhofer IESE

Your notes:

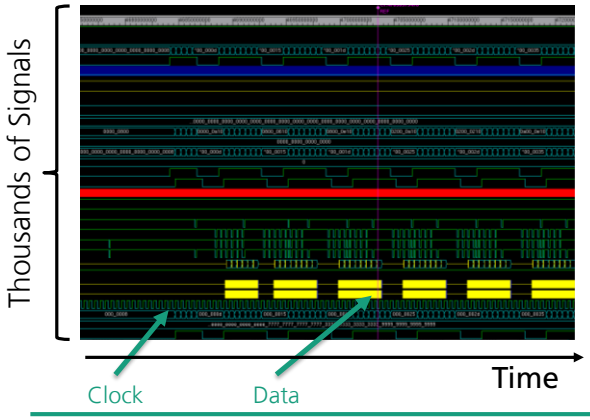
Accuracy vs. Speed Trade-Off

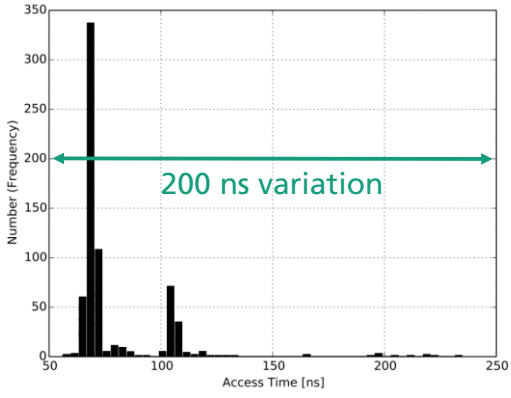
E.g. RTL Simulations:

- VHDL / Verilog / SystemC
- Very accurate
- Very very very … slow
- Inflexible

E.g. System Level Simulations:

- Fast
- Large flexibility
- Inaccurate





47

Fraunhofer IESE

Your notes:

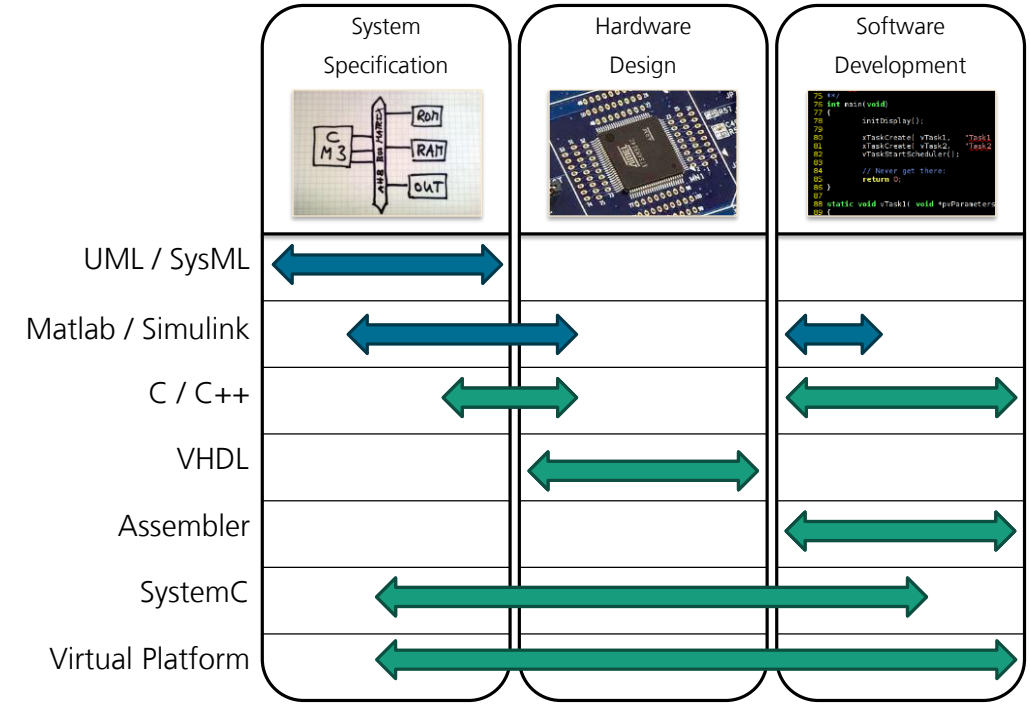
Keys to Make Simulations Fast

- Be certain about what you want to model
- Ask you, which details are really important?
- Technical Aspects:
  - Saving time simulation, events and clocks (simulate only important events!)
  - Avoid moving large amounts of data
  - Avoid simulation context switches i.e. limit the number of calls to `wait()`
  - Exploit compiler optimizations
  - Keep native data types (instead of 17-bit signal)
  - Reduce unnecessary control flow (e.g. by using polymorphism)
  - Avoid `print`, `cout`, logs, `string` processing (e.g. `std::map<string,…>`)

Fraunhofer IESE

Your notes:

Different Modelling Languages



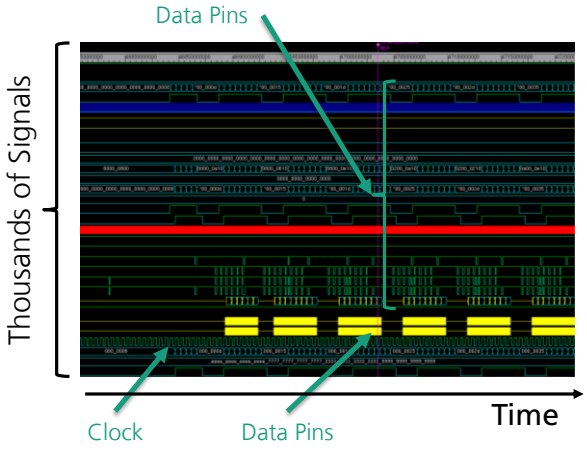
49

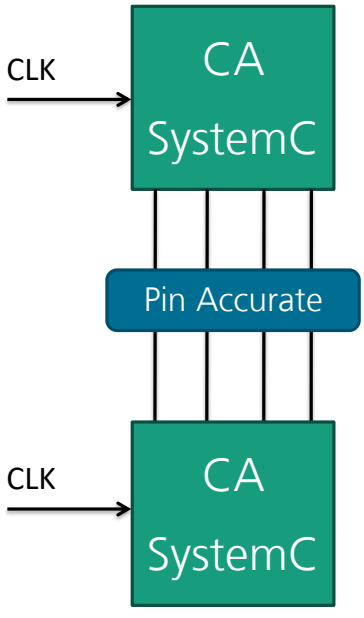
Fraunhofer IESE

To build a virtual prototype it is necessary to choose a framework, which covers the fields of system specification, hardware design and software development like SystemC, which will be discussed in this lecture.

Your notes:

Lookout: Cycle Accurate Simulation





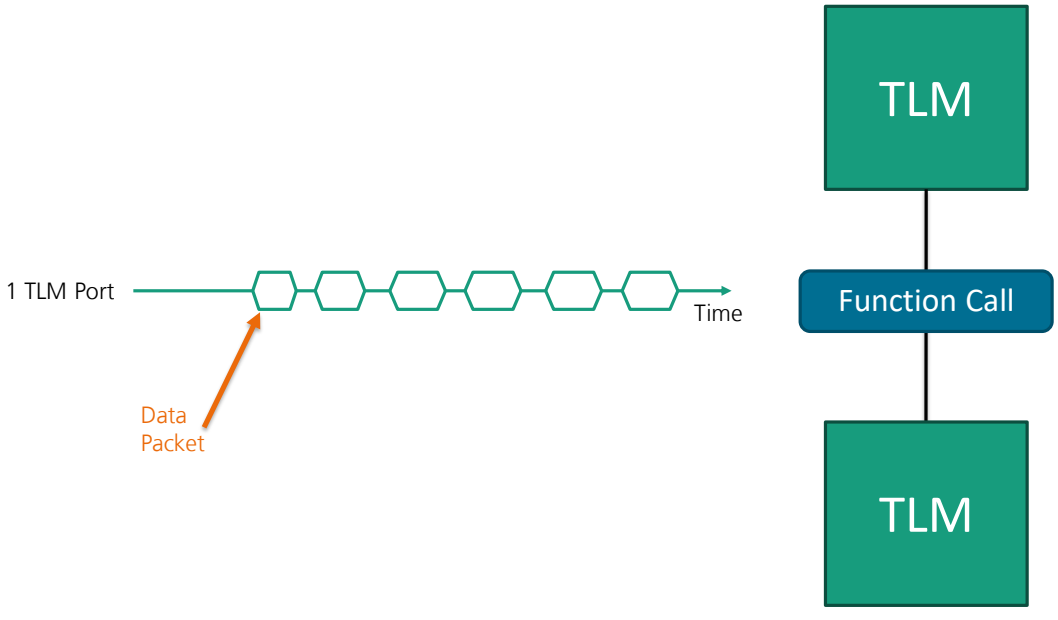
Simulate every event!!

Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Your notes:

Lookout: Transaction Level Modeling (TLM)

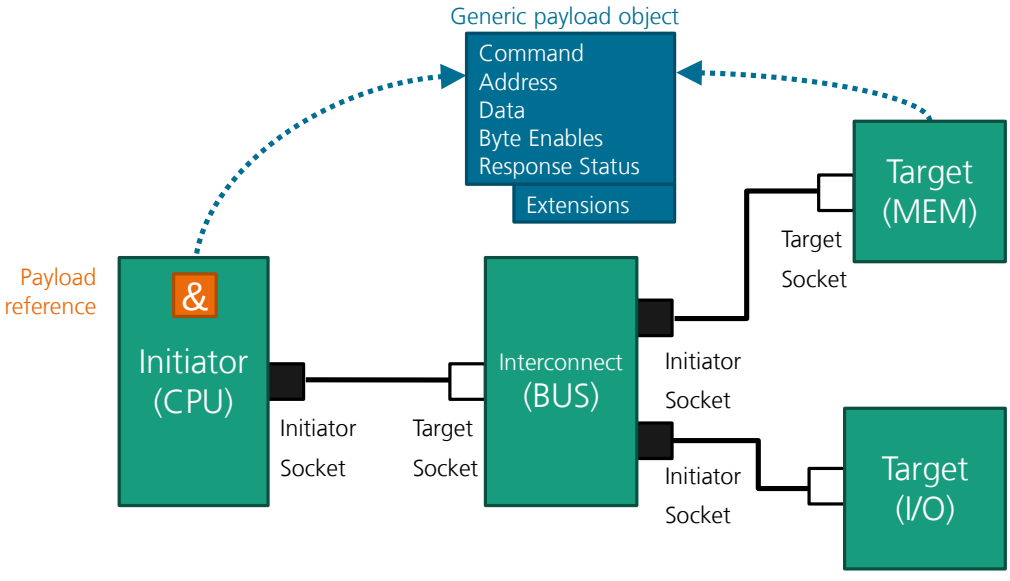


100-10,000 X faster simulation!

Fraunhofer IESE

Your notes:

Lookout: Transaction Level Modeling (TLM)



Fraunhofer IESE

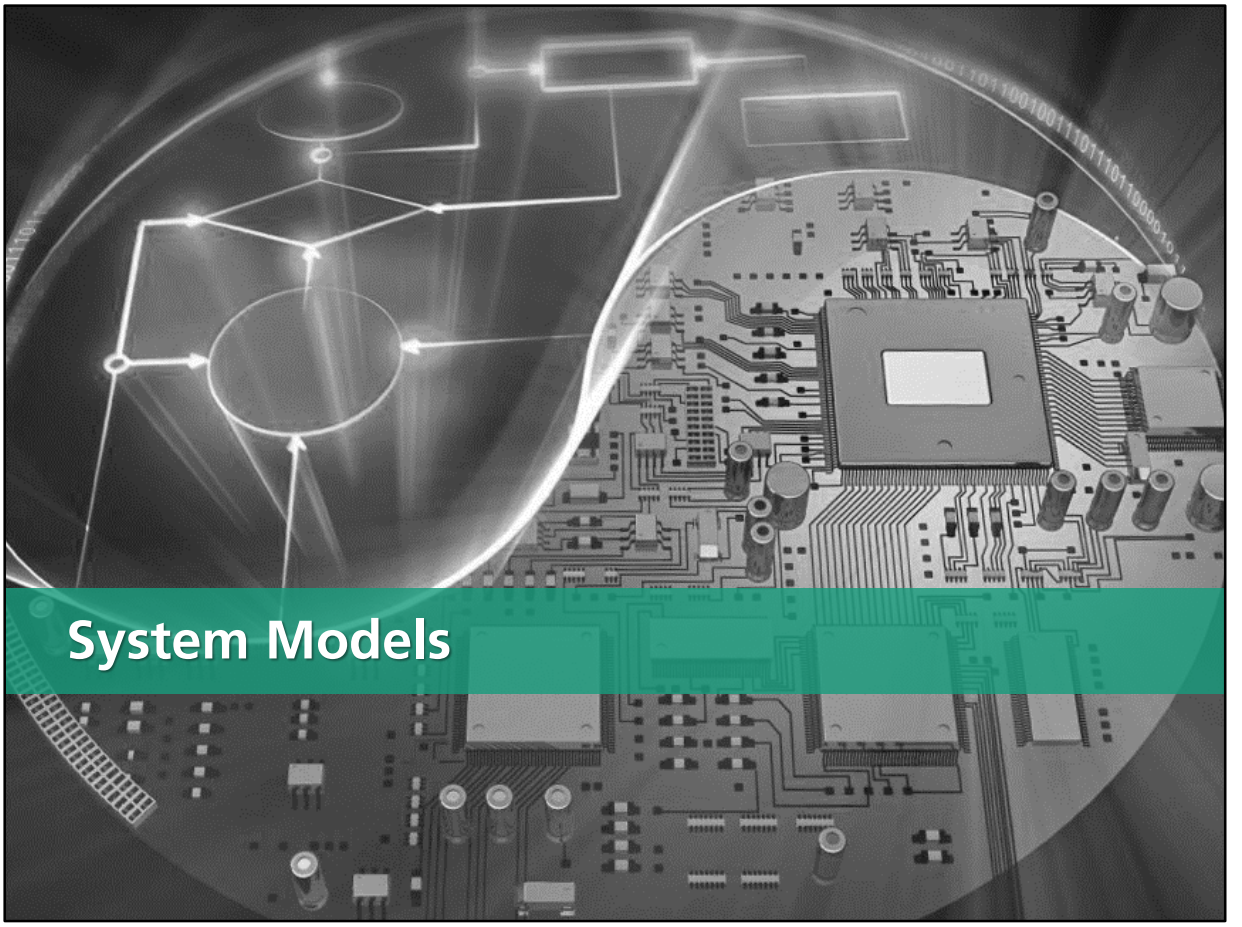
Your notes:

SystemC and Virtual Prototyping

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

Fraunhofer IESE

Your notes:



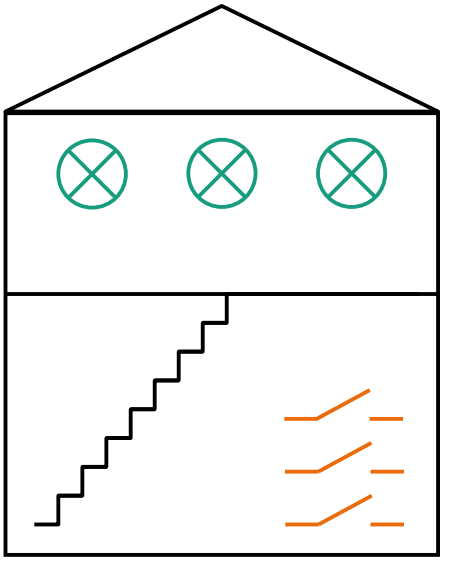
Your notes:

Test:

Models …

… *enable … thinking*

… *hamper … perception*



Fraunhofer IESE

Your notes:

Perception vs. Reality

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

4

Fraunhofer IESE

Mental models help people make sense of the world — to interpret their environment and understand themselves. Mental models include categories, concepts, identities, prototypes, stereotypes, causal narratives, and worldviews. Individuals do not respond to objective experience but to their mental representations of experience. In constructing their mental representations, people use interpretive frames provided by mental models. People may have access to multiple and conflicting mental models. Context can activate a particular mental model. Using a different mental model can change the individual's mental representation of the world around him.

Your notes:

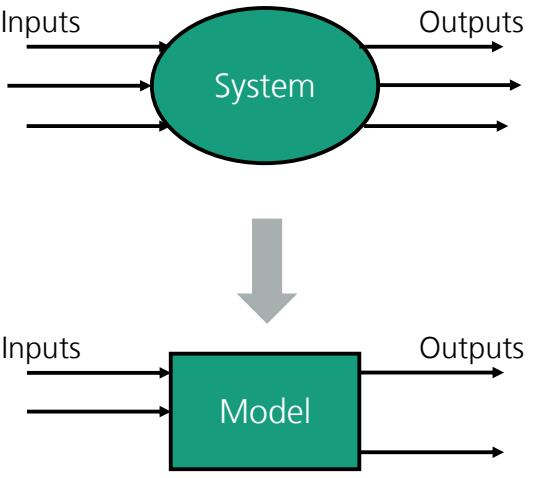
System and Model

- 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.

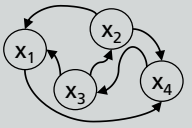
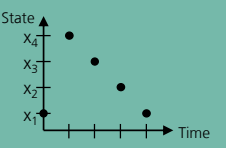
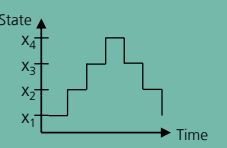
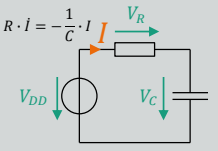
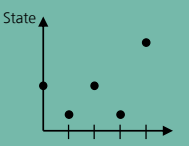
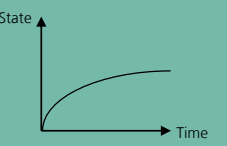
  Describes how the system works



Fraunhofer IESE

Your notes:

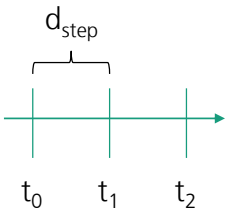
Time and States

	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) 		

Fraunhofer IESE

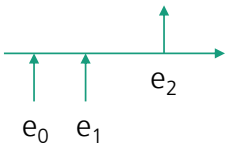
Your notes:

Time Simulation Concepts



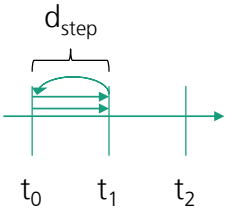
Discrete Time Simulation:

- 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

- Discrete time (DT) MOCCs split the execution into discrete time steps of constant or variable size. They support for example the implementation of discre-tized physics models and control algorithms. Input ports under control of a DT MOCC store one value that may change between time steps, but that is kept constant during time steps of a simulation. Output ports communicate simula-tion results at the end of a time step. The MOCC executes controlled compo-nents in any order. The duration of a time step dstep reflects the granularity of the simulation; shorter time steps yield higher accuracy, but also require more frequent calculations and more computation time. Larger time steps require less calculations but yield a higher discretization error.
- Discrete event (DE) simulation models implement event based communication and the processing of simulation events. Events may occur at any time and are not bound to time steps. Their execution is ordered based on expiration times. This ensures that the simulation time in discrete simulation models continuously increases, but requires overhead for the necessary sorting of event. Simulation components receive events via their input ports. Discrete event simulations usually use a two-dimensional time.
- Continuous time (CT) simulation approximates the continuous behavior of re-al-world physics. Changes to individual elements have immediate impacts to dependent elements. For example, when a spring is extended, it immediately applies force to both ends. Resembling this behavior in a simulation is tricky, as necessary discretization of the simulation prior to solving yields a simulation error. Every CT simulation is controlled by a solver that evaluates one simula-tion component after another in discrete time steps. After simulating one component, it copies output values to the inputs of dependent components. Solvers control simulation errors by iterating the same simulation step until the simulation error falls below an acceptable threshold.

MOC Support in SystemC

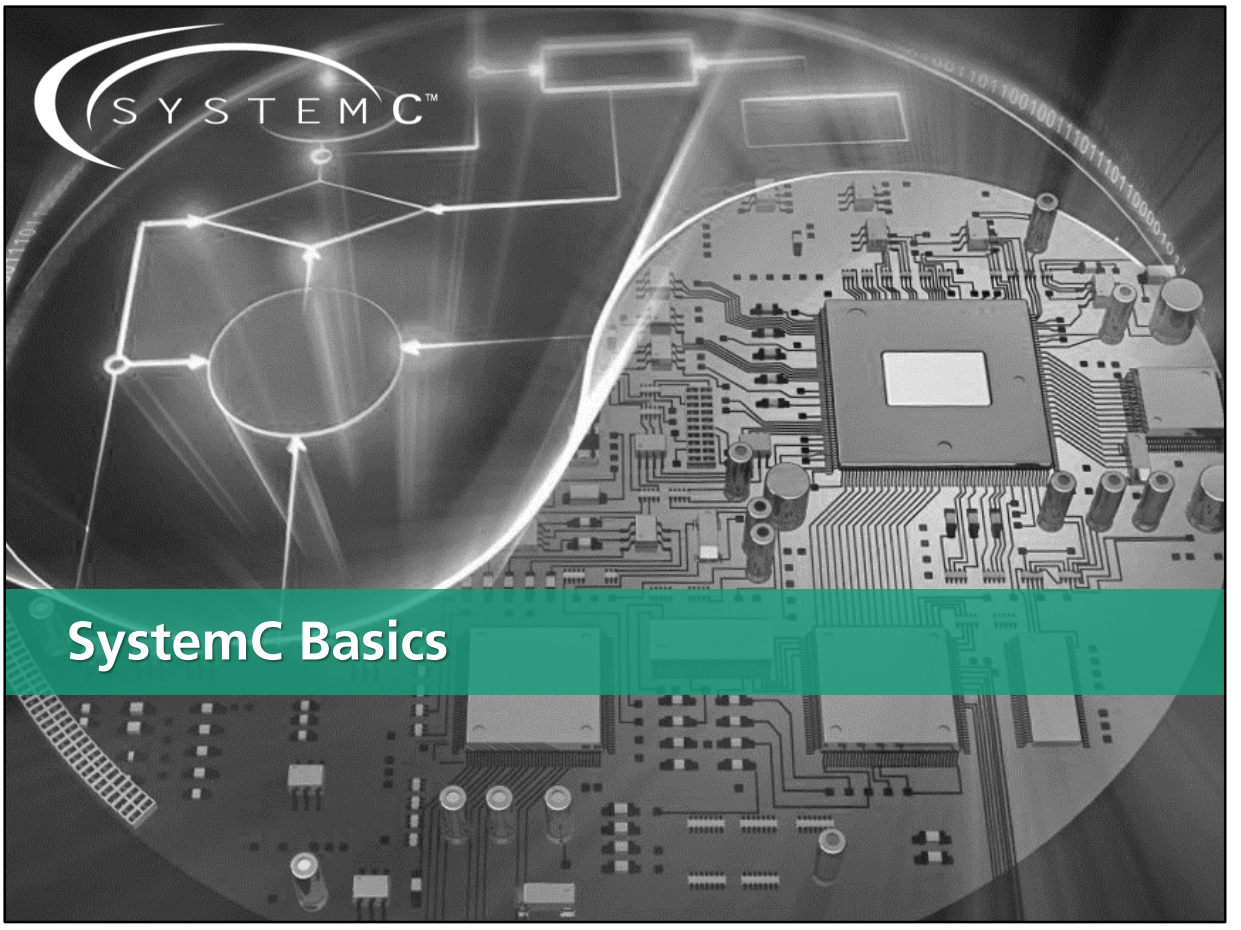
- **Discrete Event** as used for:
  - RTL Hardware Modeling
  - State Machines
  - Network Modeling (e.g. stochastic or "waiting room" models)
  - Transaction Level Modeling
- Continous Time with AMS-Extension
- Kahn Process Networks
- Static Multi-rate Data-flow
- Dynamic Multi-rate Data-flow
- Communicating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

Fraunhofer IESE

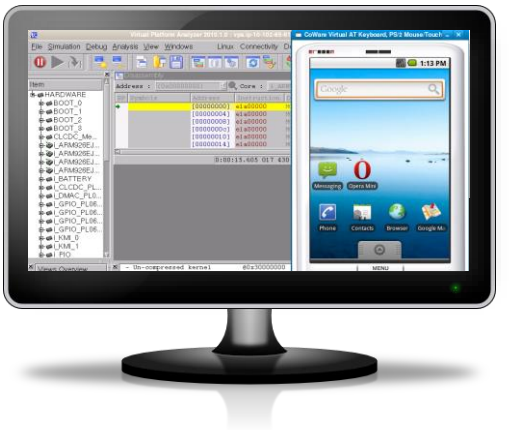
Your notes:



Your notes:

Move to Virtuality?

Everything is in the Developer's Desktop

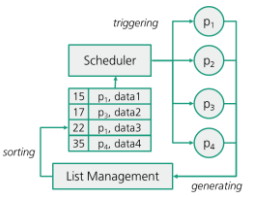


Fraunhofer IESE

Your notes:

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

Your notes:

Install SystemC on your Private Machine

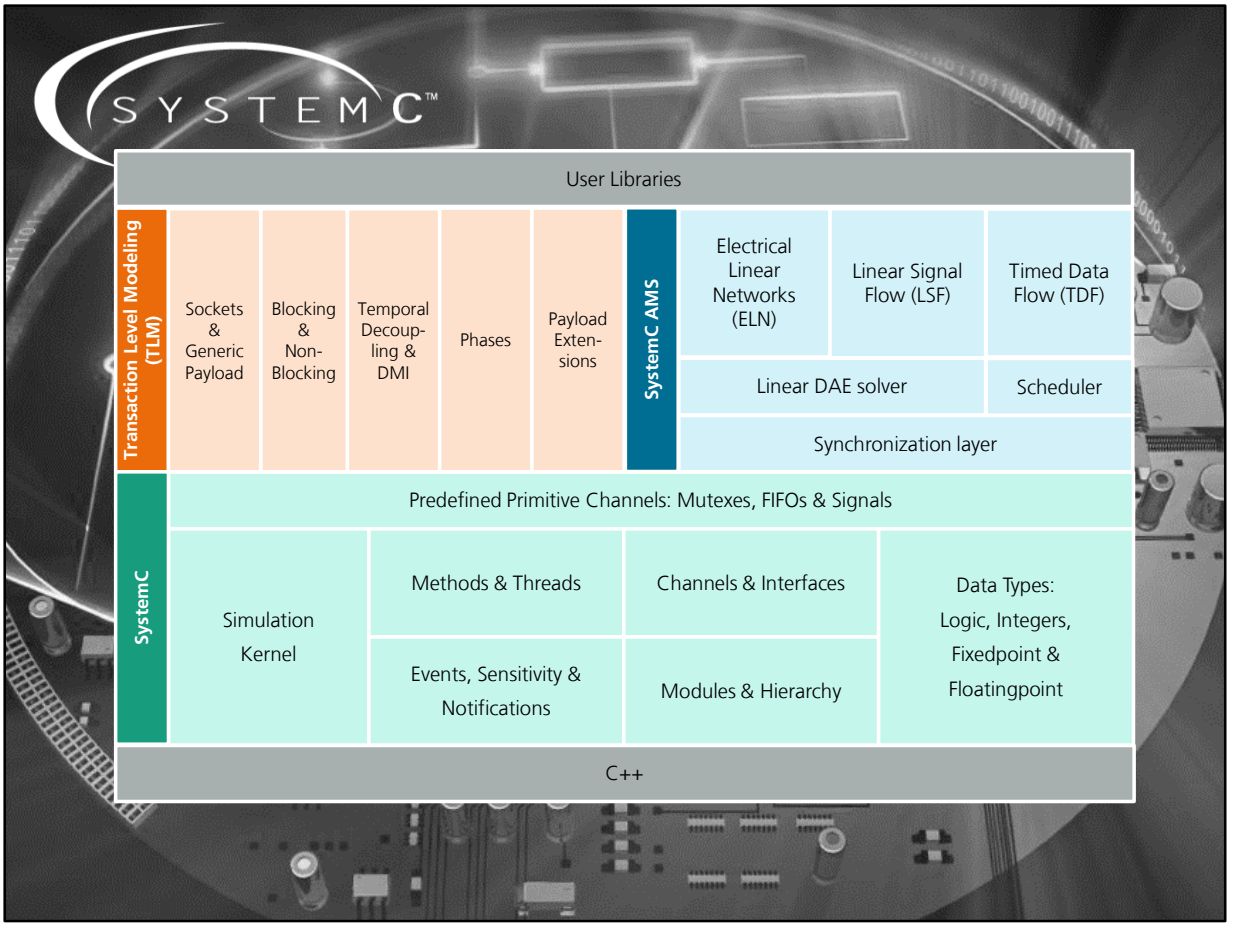
For Example on Ubuntu or Debian like Linux distributions

```
$ 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
```

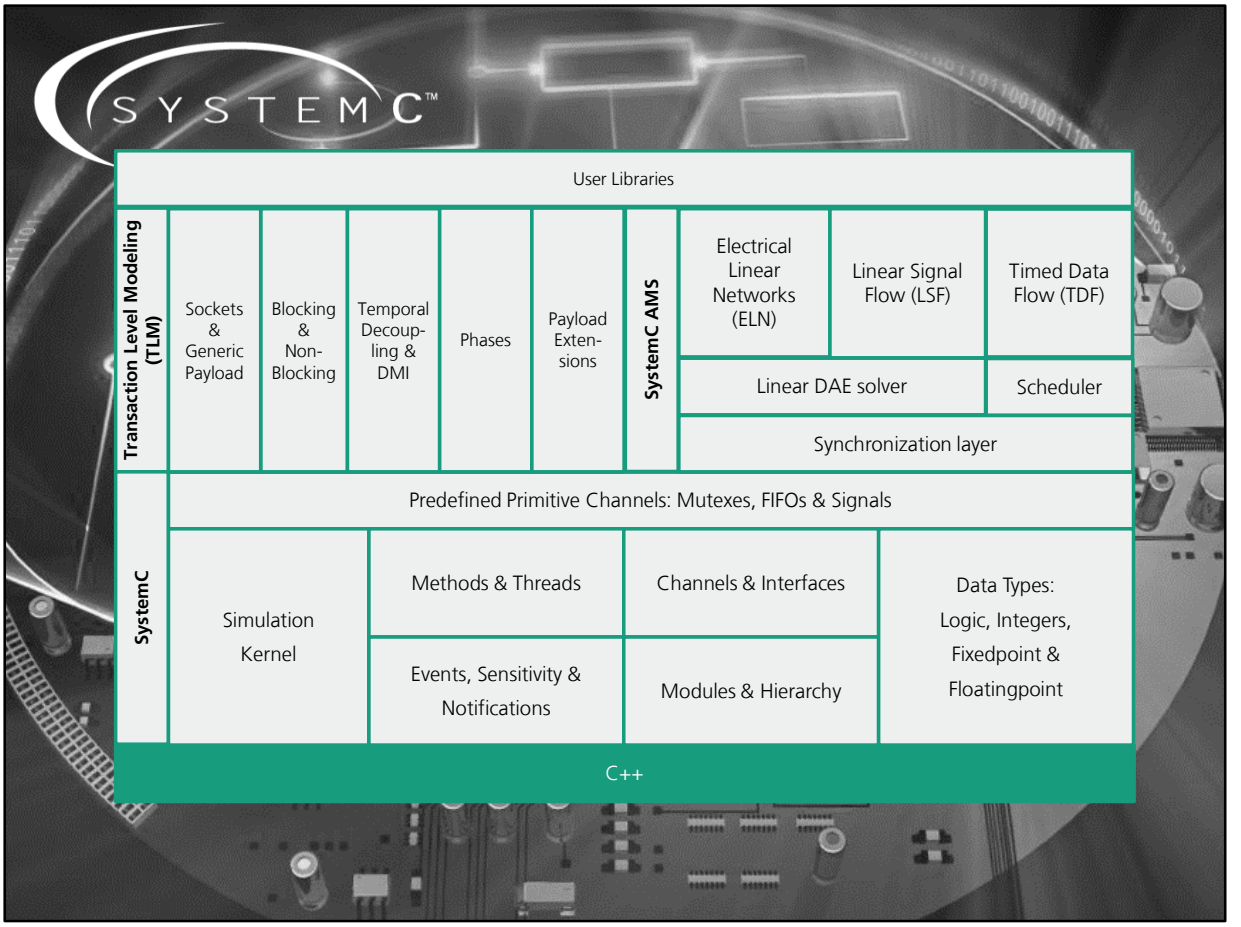
Get script on GitHub:
https://github.com/tukl-msd/SCVP.artifacts/blob/master/install_systemc.sh

Fraunhofer IESE

Your notes:


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++

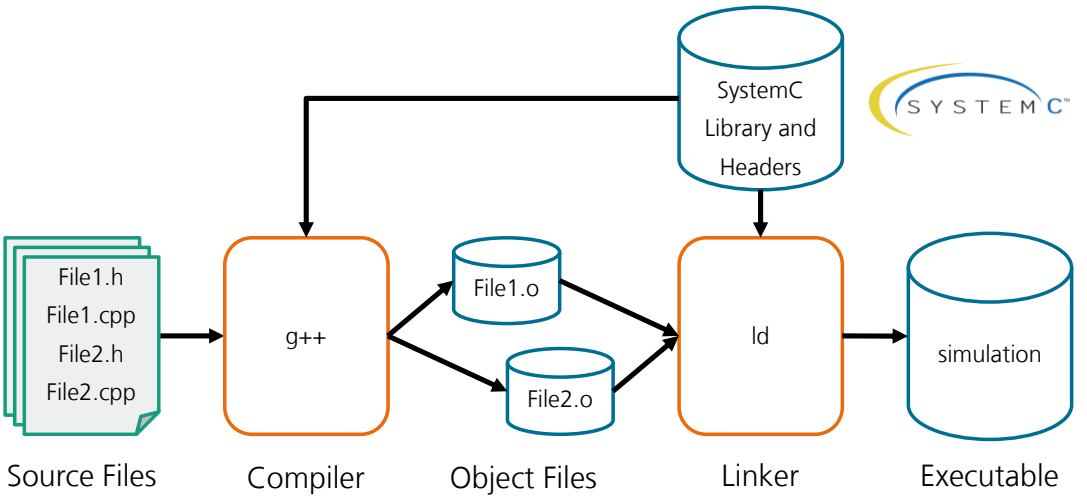
Your notes:


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++

Your notes:

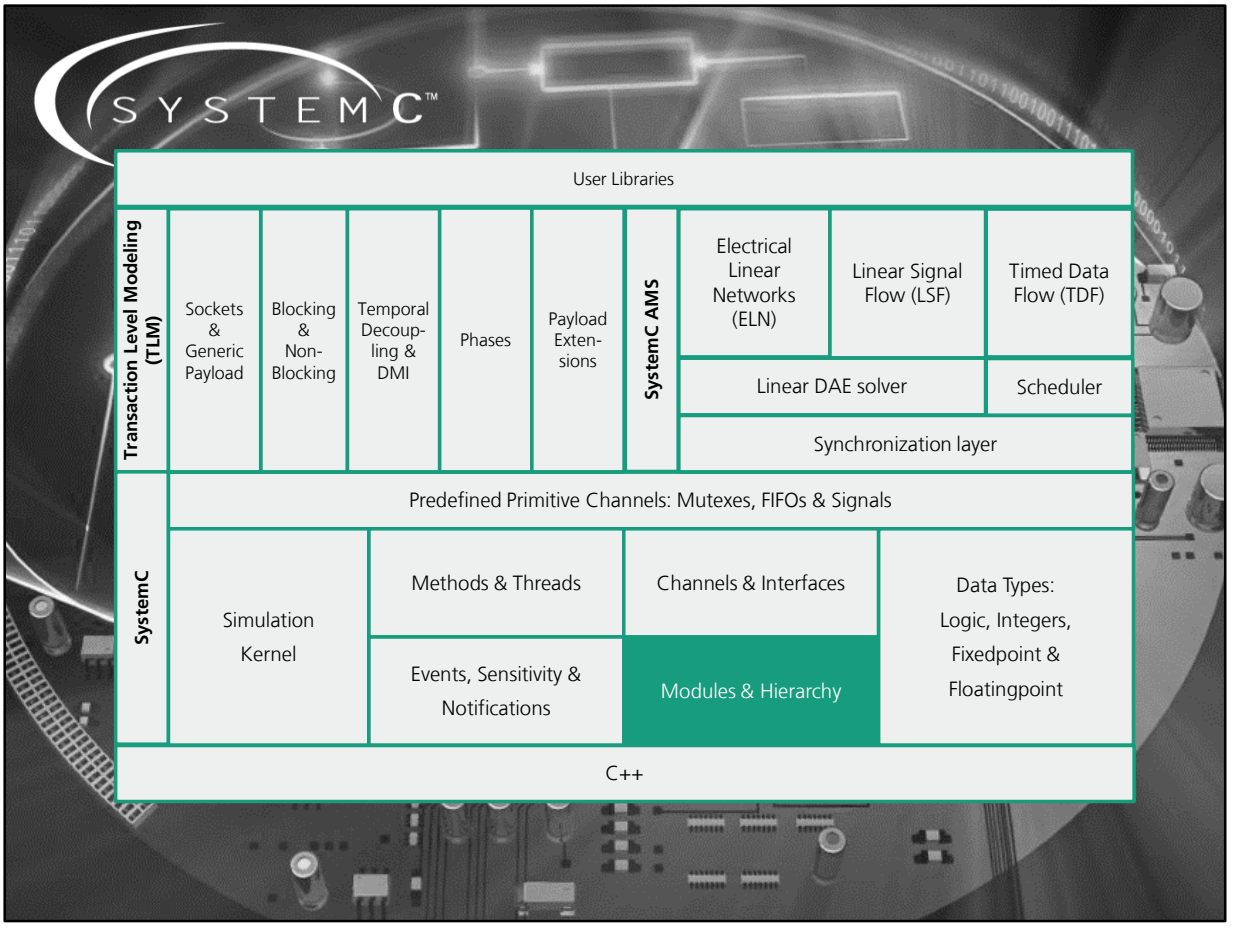
SystemC Compilation Flow

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



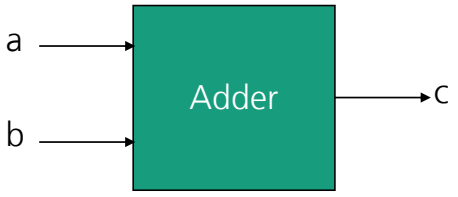
Fraunhofer IESE

Your notes:


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++

Your notes:

SystemC Basic Example



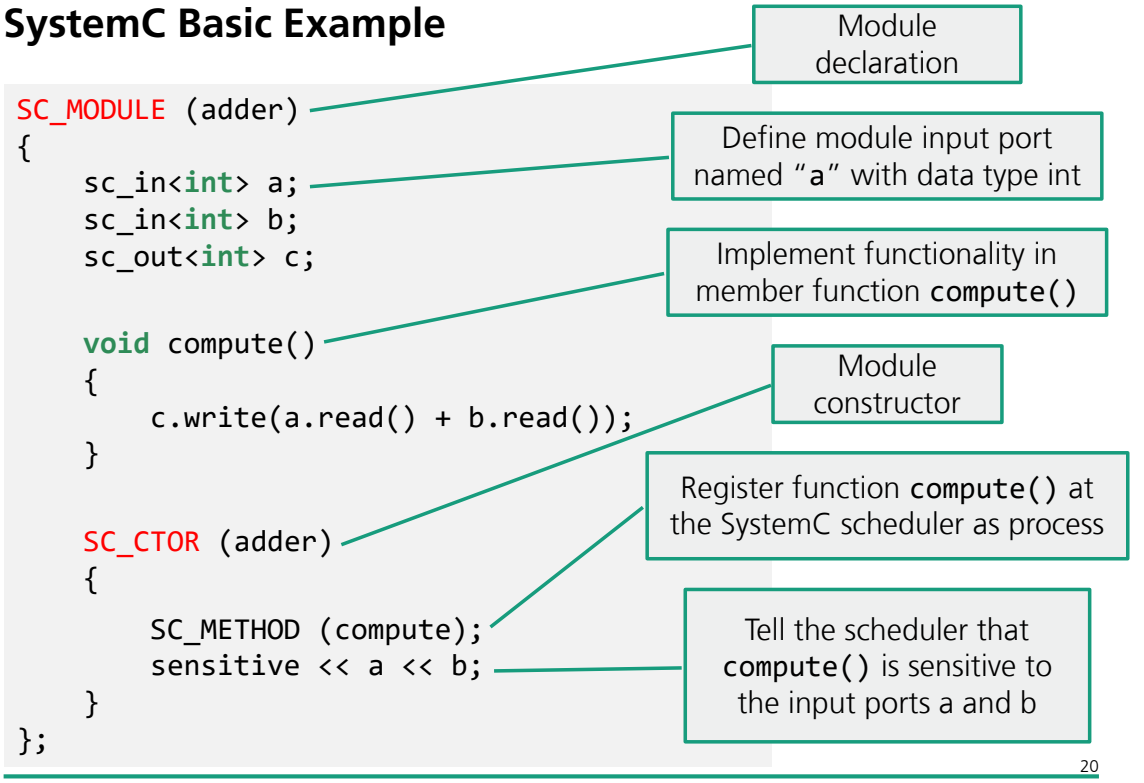
Remember an adder in VHDL:

```
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;
```


Fraunhofer IESE

Your notes:

SystemC Basic Example



20

Fraunhofer IESE

Your notes:

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:

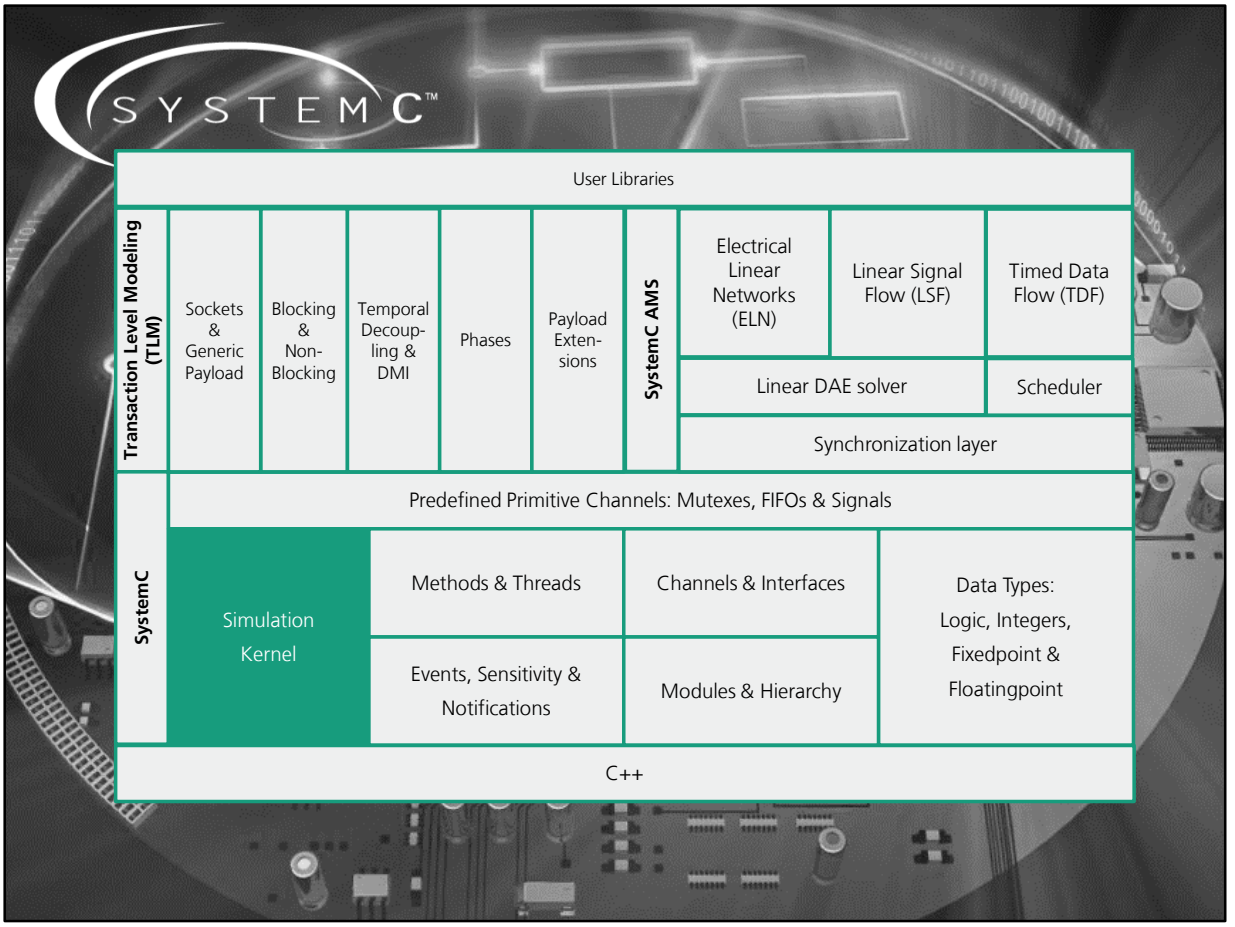
```
typedef XYZ SC_CURRENT_USER_MODULE
```

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

Your notes:

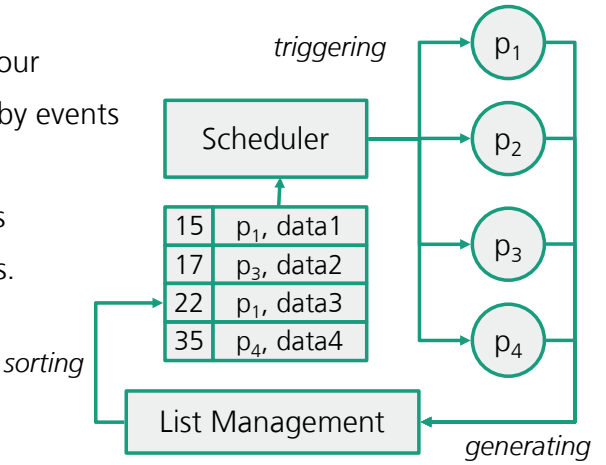

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:

Discrete Event Models (DEM) – General Concept

Evaluation of state changes only at occurrence of events!

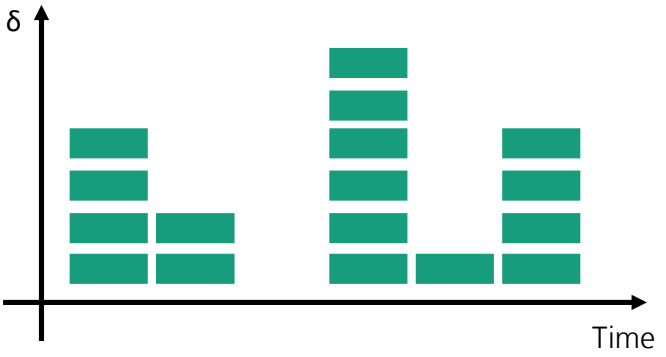
- Process describes functional behaviour
- Execution of processes is triggered by events
- Processes are deterministic
- Processes may generate new events
- Events are sorted w.r.t. time stamps.



Fraunhofer IESE

Your notes:

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

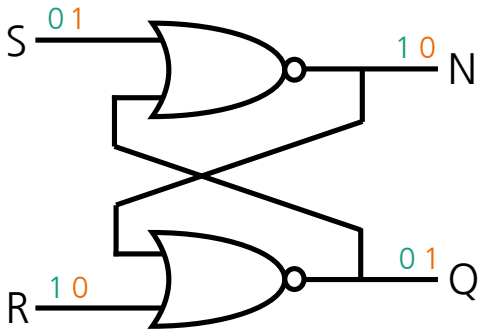


- The δ-Delay enables the simulation of concurrency in a sequential simulator
- The δ-Delay is an infinitesimally small abstract time unit
- The δ-Delay guarantees a deterministic signal assignment
- The δ-Delay is used, if a statement with 0 ns or SC_ZERO_TIME is called.

Fraunhofer IESE

Your notes:

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

26

Fraunhofer IESE

The internal functionality of the RS latch is described by the following two statements:

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

In general, the signal on the left side (*) of the = depends on all the signals appearing on the right side. Therefore, the output of Q depends on N and the output of N depends on Q. If a signal depends on another signal, which has changed previously, then the expression in the signal assignment is re-evaluated in a so called δ-cycle. If the result of the evaluation is different than the current value of the signal, an event will be scheduled (added to the list of events to be processed) to update the signal with the new value and re-evaluate dependent equations. For example, if a change occurs on R or N, then the nor operator is evaluated, and if the result (Q^*) is different than the current value of Q, an event will be scheduled in order to update Q and re-evaluate the equation $N^* = \overline{S \vee Q}$. For the example in the slides: at simulation time 0ns the signals are set to $S = 0$, $R = 1$, $Q = 0$, and $N = 1$. At the time of 10 ns two values change to $S = 1$ and $R = 0$. Since we have dependencies we have to evaluate the process again:

$$Q^* = \overline{R \vee N} = \overline{0 \vee 1} = \overline{1} = 0$$
$$\boldsymbol{N^* = \overline{S \vee Q} = \overline{1 \vee 0} = \overline{1} = 0}$$

Apparently, the value of N has changed from 1 to 0. Therefore, we have to re-evaluate again the process in a next round (δ-cycle), in particular the equation $Q^* = \overline{R \vee N}$ because it depends on N:

$$\boldsymbol{Q^* = \overline{R \vee N} = \overline{0 \vee 0} = \overline{0} = 1}$$
$$N^* = \overline{S \vee Q} = \overline{1 \vee 0} = \overline{1} = 0$$

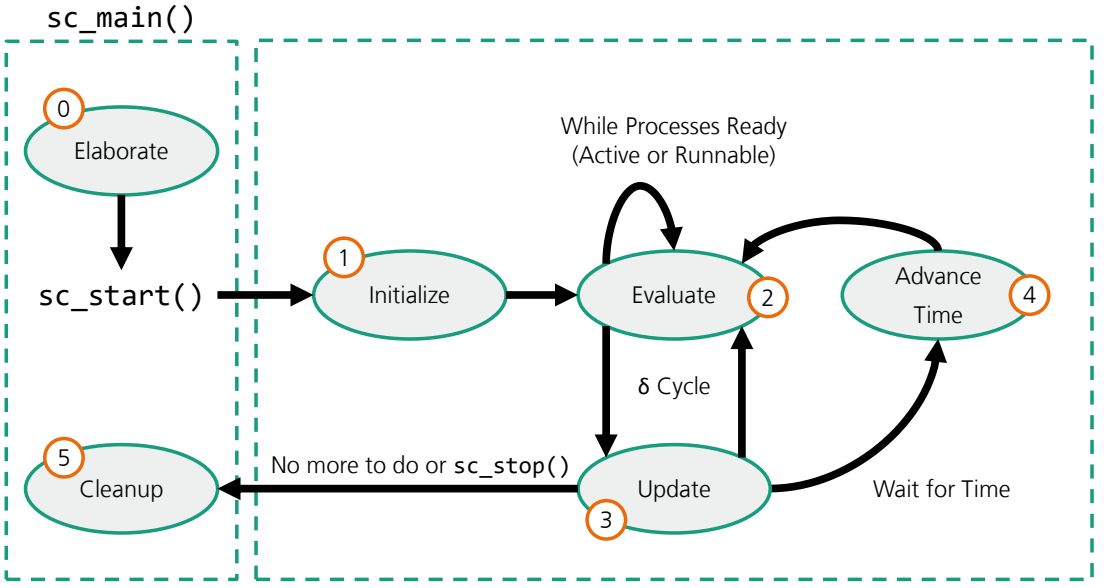
Now, the value of Q has changed from 0 to 1. Therefore, we have to re-evaluate again the process (δ-cycle):

$$Q^* = \overline{R \vee N} = \overline{0 \vee 0} = \overline{0} = 1$$
$$N^* = \overline{S \vee Q} = \overline{1 \vee 1} = \overline{1} = 0$$

Now, there is no value change and therefore we are finished with δ-cycling. All the signals have reached their stable

values.

SystemC Simulation Kernel



27

Fraunhofer IESE

Your notes:

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

Your notes:

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

Your notes:

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

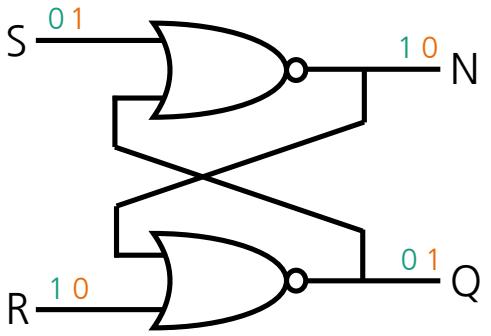
4 ***Advance Time***: Processes sensitive to events in the event queue with the smallest time are marked for execution and the scheduler proceeds to the evaluation phase 2 and thus, the simulation time is advanced. If there are no events in the event queue the simulation is finished. Then the scheduler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

Your notes:

Remember: 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

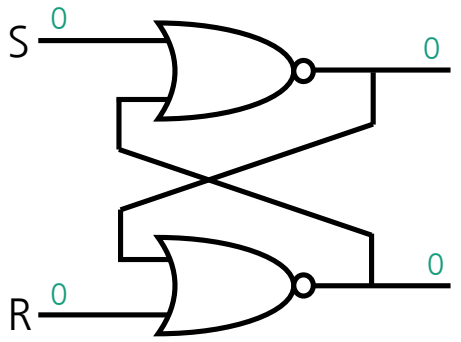
31

Fraunhofer IESE

See explanation on System Model Chapter …

Your notes:

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

Your notes:

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.:

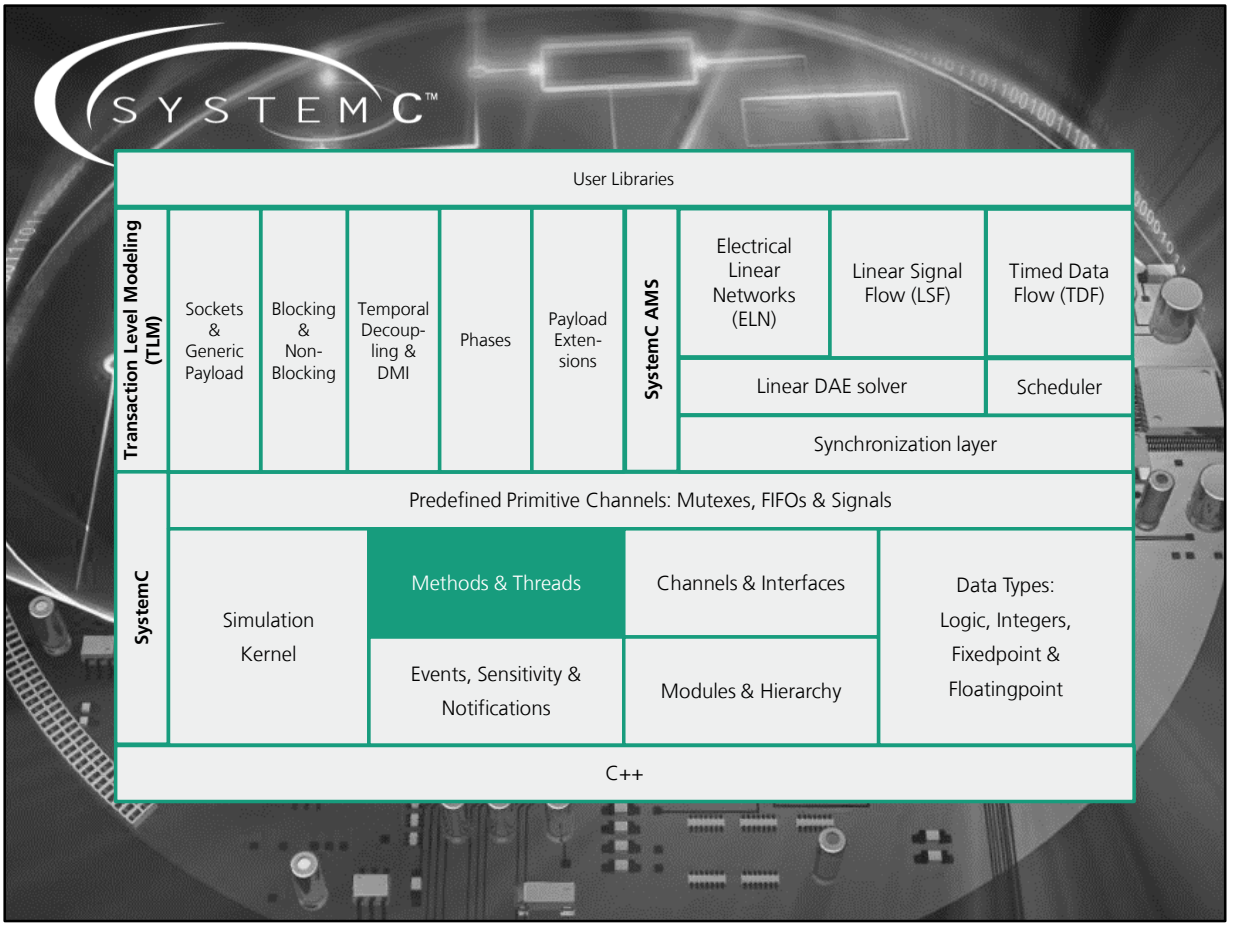
```
void process() // sc_signal<int> C=3 D=4
{
    C = D;
    D = C;
}
```

- 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

Fraunhofer IESE

Your notes:


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++

Your notes:

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

36

Fraunhofer IESE

Your notes:

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

Your notes:

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!

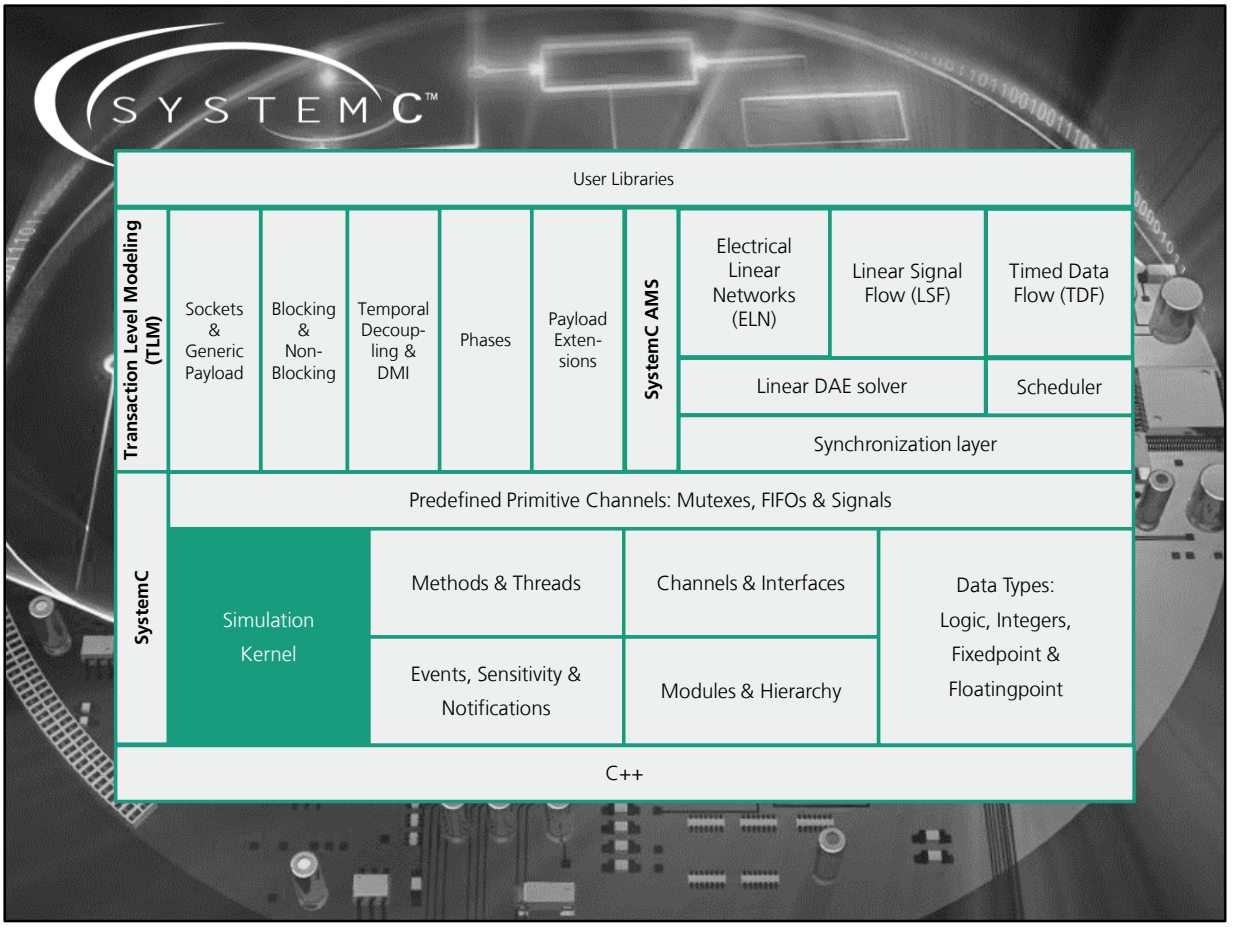
Try code on github:
https://github.com/TUK-SCVP/SCVP_artifacts/tree/master/thread_example

```
#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
        }
    }
};
```

38

Fraunhofer IESE

Your notes:


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++

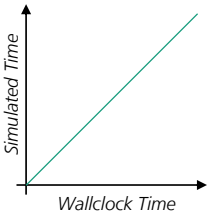
Your notes:

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.

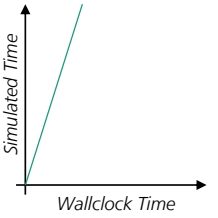
"Real-Time" (HiL)

WCT = ST

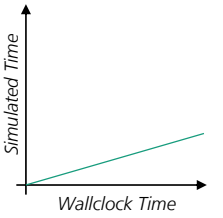


"As Fast as Possible" (vHiL)

WCT < ST



WCT > ST



Fraunhofer IESE

Your notes:

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

Your notes:

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();
...
```

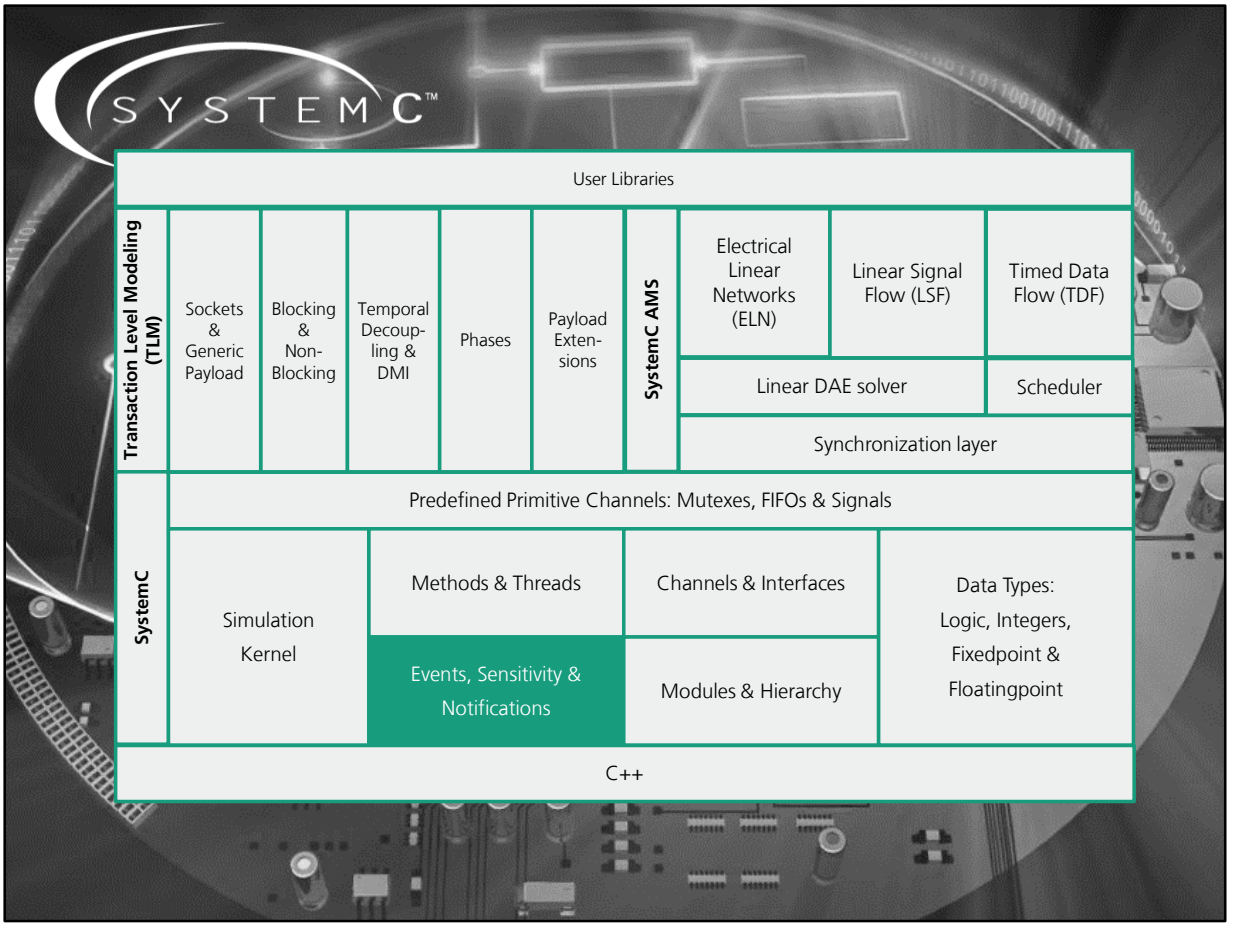
Simulation can run until there are no events, to a limited time, or unitl a call of `sc_stop()` in a process

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

Your notes:


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++

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:

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 overwritten temporarily.

Fraunhofer IESE

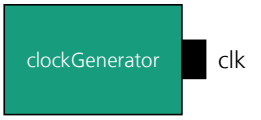
Your notes:

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

Your notes:

SystemC Events: Sensitivity

Static Sensitivity

Dynamic 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;
  }
};
```

THREAD

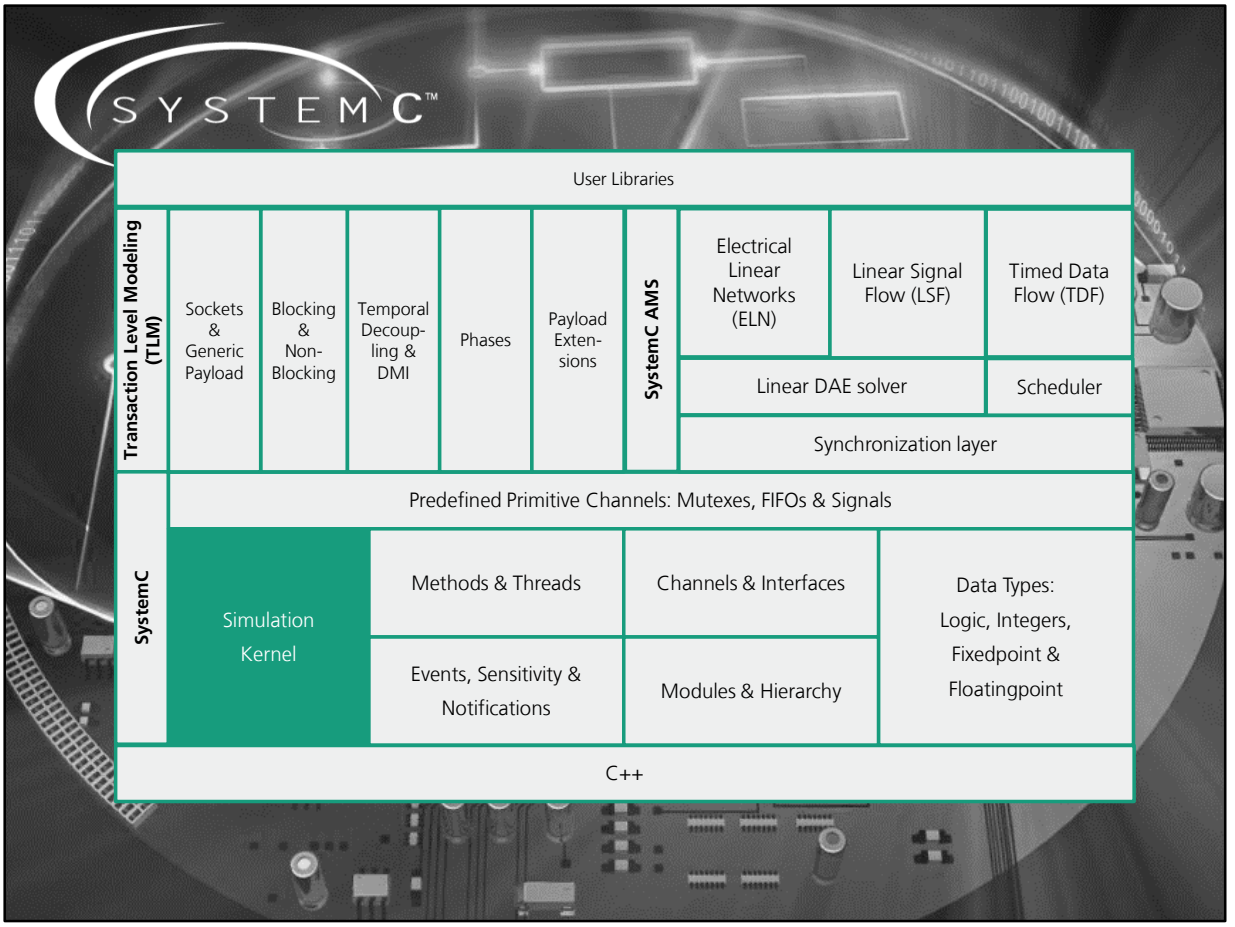
```
SC_MODULE (Module)
{
  sc_in<int> a;

  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);
  }
};
```


Fraunhofer IESE

Your notes:

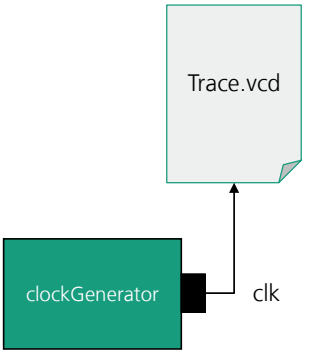

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:

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;
}
```

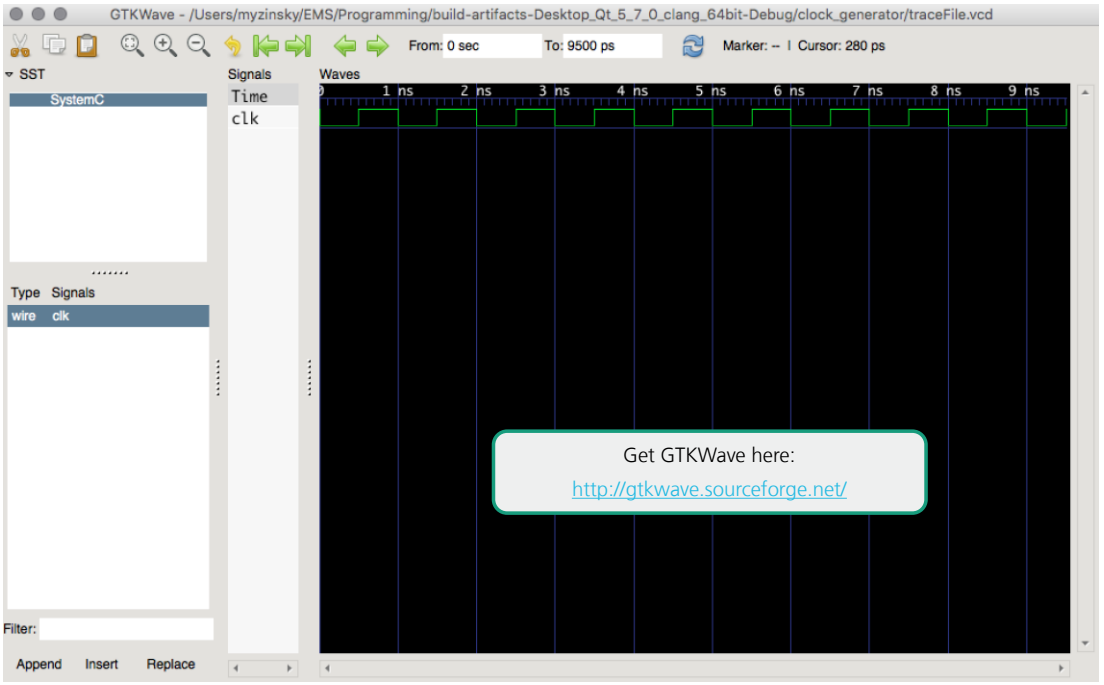
- Like VHDL or Verilog, SystemC allows the non-intrusive recording of siganls into a waveform vcd file
- cout is printed in every delta cycle -> confusing



Fraunhofer IESE

Your notes:

SystemC Waveform Tracing



Fraunhofer IESE

Your notes:

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:

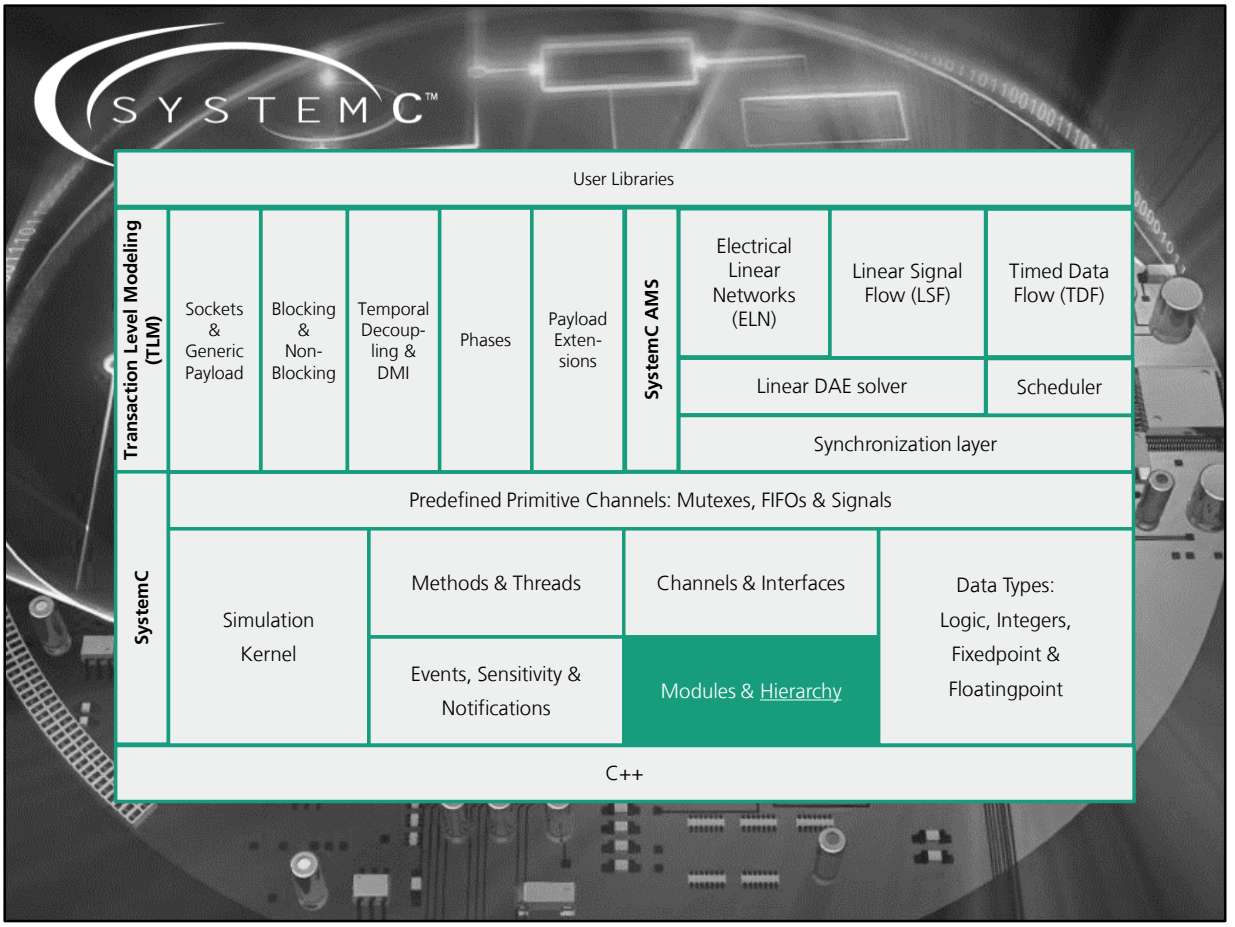
```
sc_clock clock("Clk", 10, SC_NS, 0.5, 10, SC_NS);
sc_clock clock("Clk2", sc_time(10, SC_NS));
sc_clock clock("Clk3", 10, SC_NS, 0.5);
```

- Processes can be sensitive to clocks:

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


Fraunhofer IESE

Your notes:

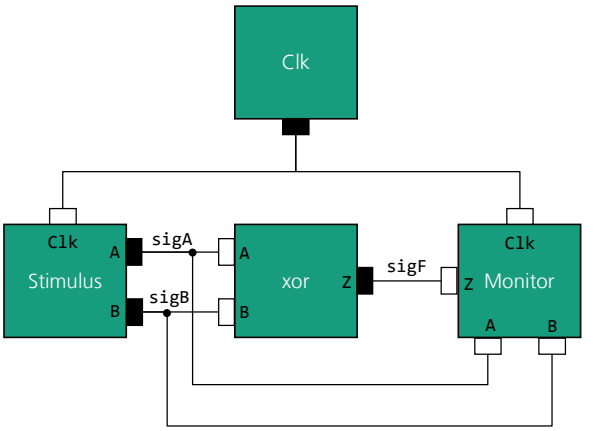

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:

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;
}
```

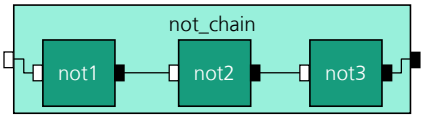
- Connecting `SC_MODULE`s in `sc_main` or in a toplevel module
- Binding of components with signals
- Keyword bind can be used or not



Fraunhofer IESE

Your notes:

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

Your notes:

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)

Fraunhofer IESE

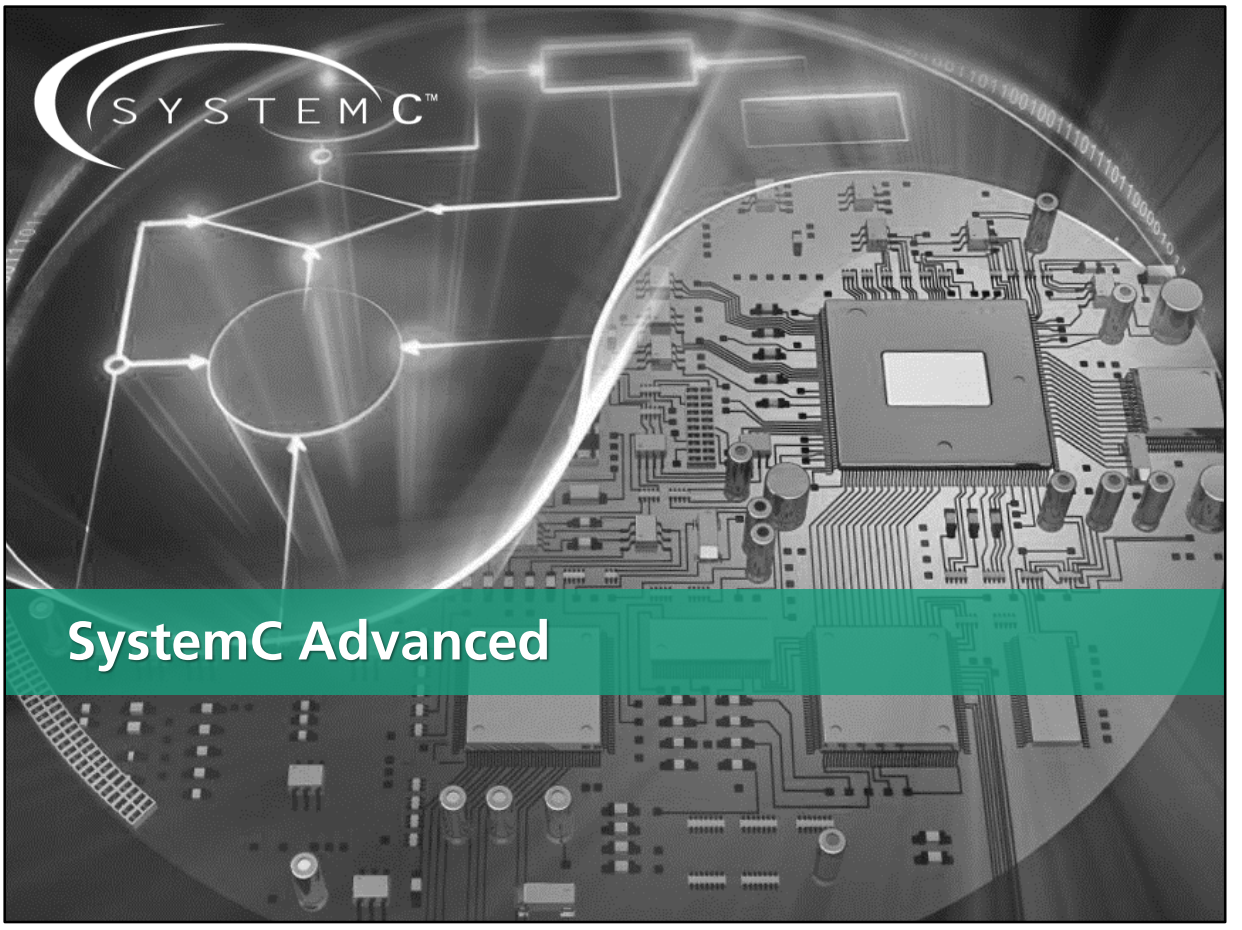
Your notes:

SystemC and Virtual Prototyping

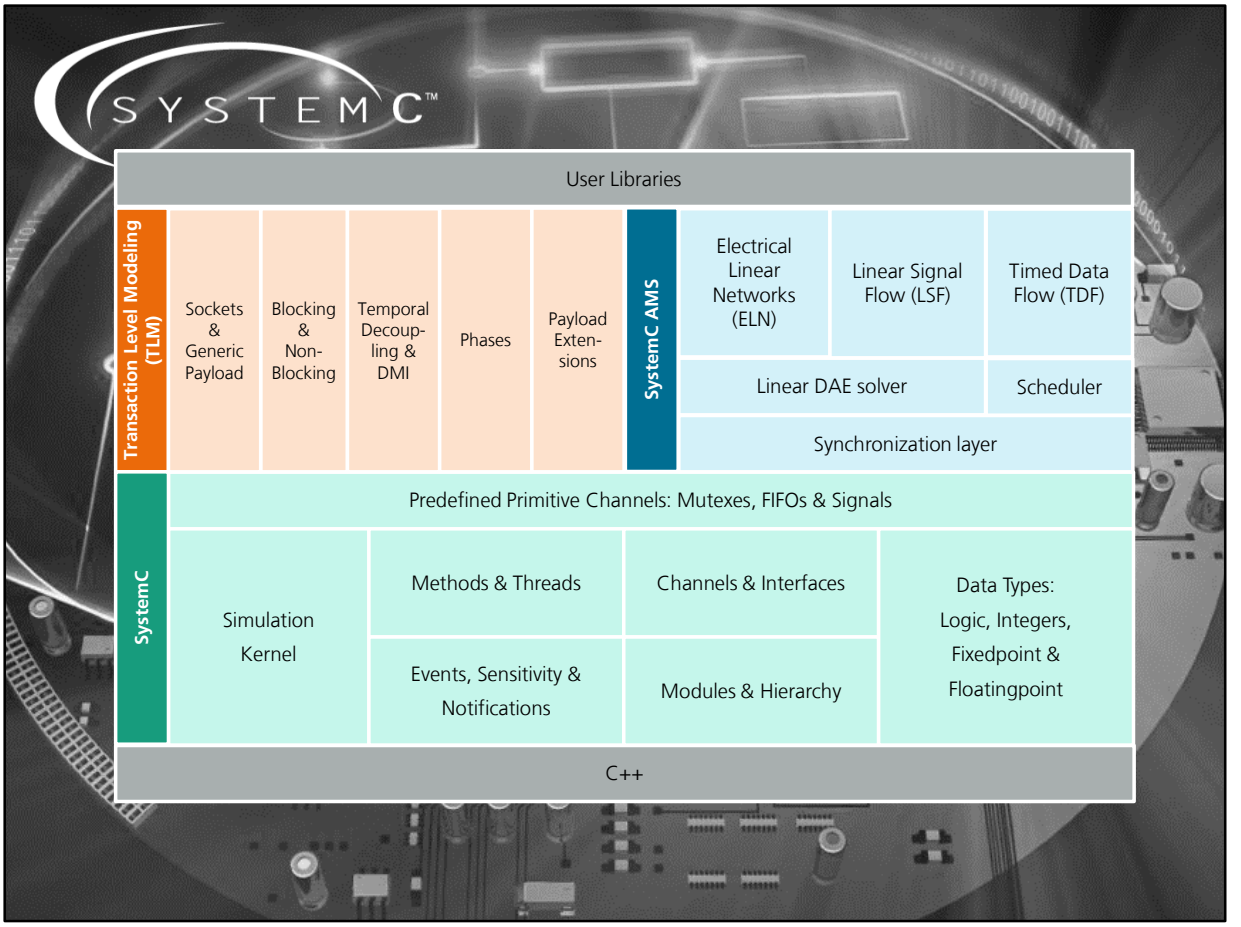
Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

Fraunhofer IESE

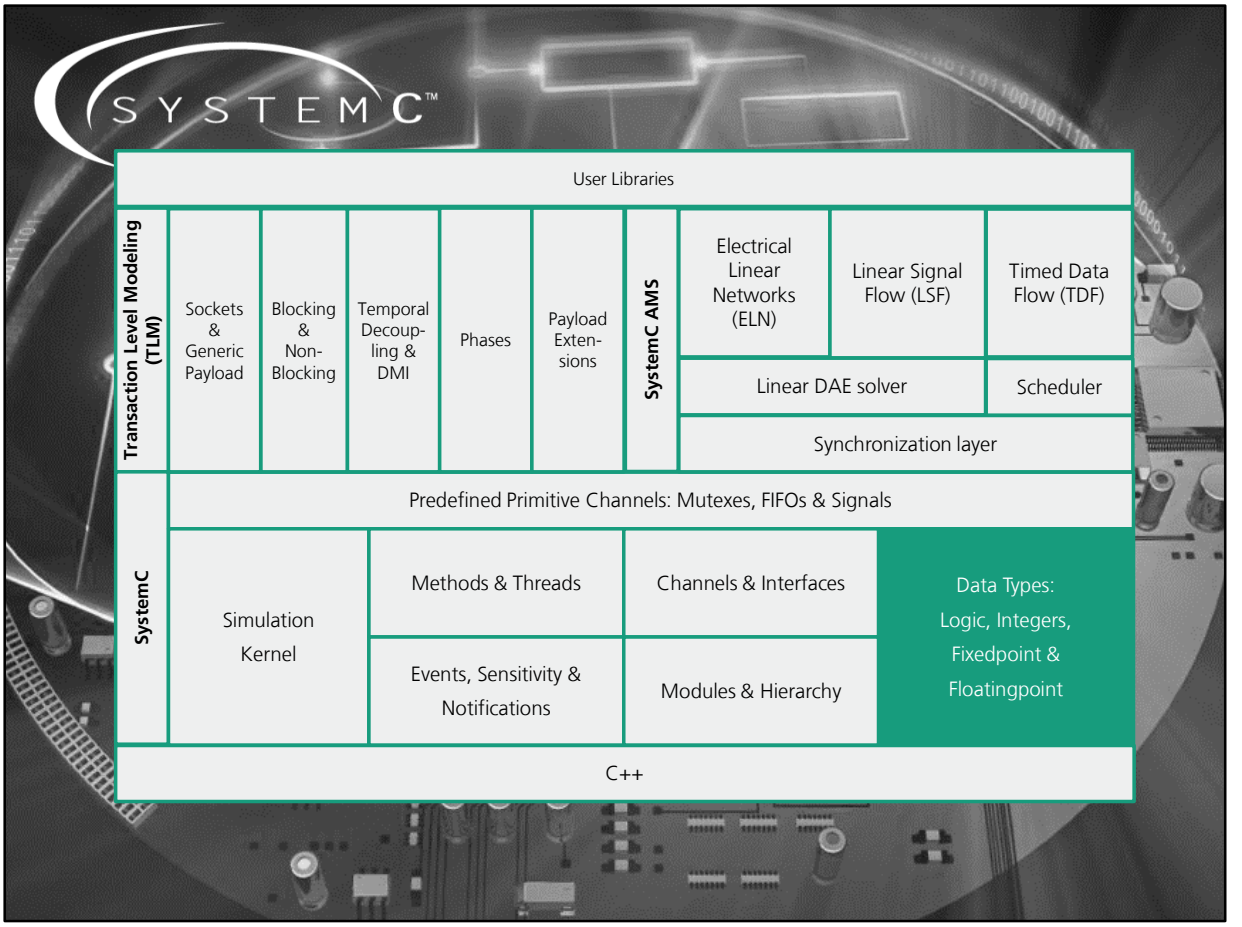
Your notes:



Your notes:


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++

Your notes:


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++

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:

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

Fraunhofer IESE

Your notes:

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

8

Fraunhofer IESE

Your notes:

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.

Fraunhofer IESE

Your notes:

Remember: Fixed Point and Two's Complement Numbers

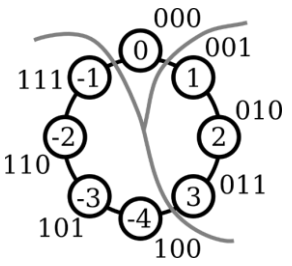
- Positive Fixed point:

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

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

- Two's Complement:

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



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

Fraunhofer IESE

Your notes:

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

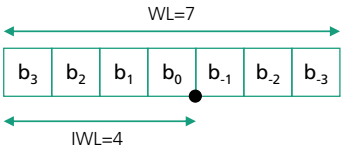
- `sc_int<W>` for signed integers and `sc_uint<W>` for unsigned integers
  - Provides efficient way to model data with specific widths (1-64)
  - When modeling numbers where data width is not an integral multiple of the simulating processor's data paths, some bit masking and shifting must be performed, which leads to an overhead in wall clock time.
- `sc_bigint<W>` and `sc_ubigint<W>`
  - Support lage data width (e.g. 512)
  - Cost of speed!

Fraunhofer IESE

Your notes:

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>:`

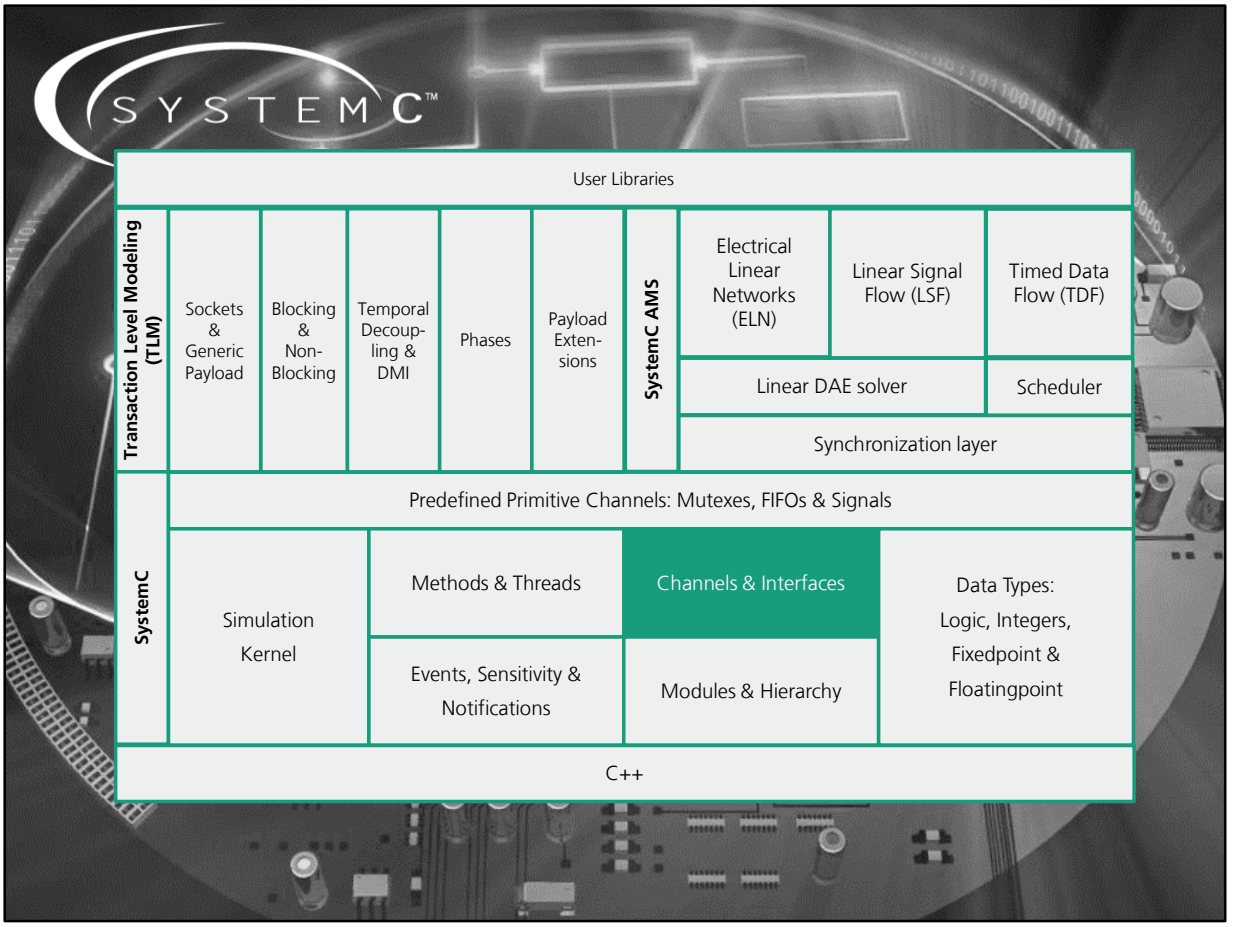


WL = Word Length
IWL = Integer WL

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

Fraunhofer IESE

Your notes:


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++

Your notes:

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

14

Fraunhofer IESE

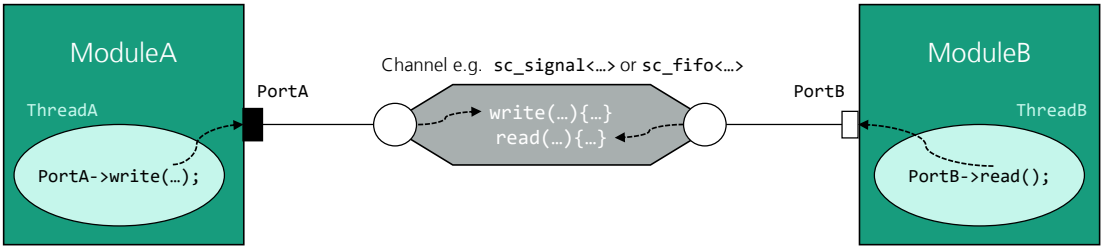
A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract. Classes containing pure virtual methods are termed "abstract" and they cannot be instantiated directly.

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*



15

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:

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

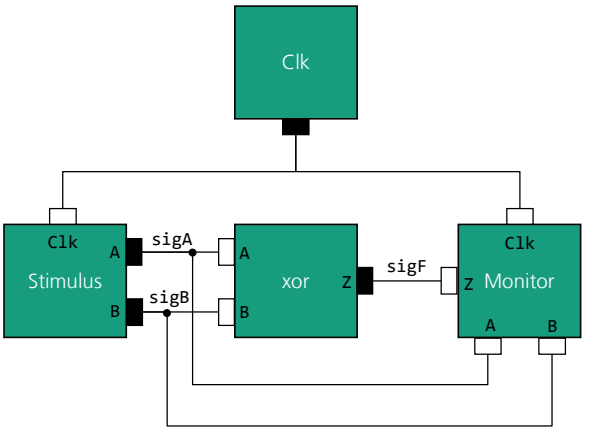
- There exist several binding policies for sc_port:
  - SC_ONE_OR_MORE_BOUND: Is the default value. The port can be bound to 1 to N channels. This implies that the port must also be bound to at least one channel.
  - SC_ZERO_OR_MORE_BOUND: The port can be bound to 0 to N channels. This implies that the 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.



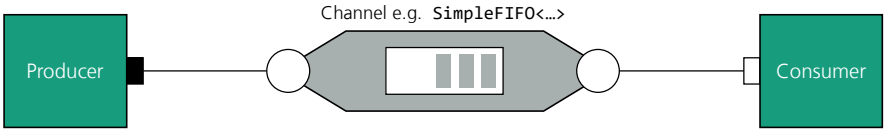
18

Fraunhofer IESE

Your notes:

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

Fraunhofer IESE

Your notes:

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

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

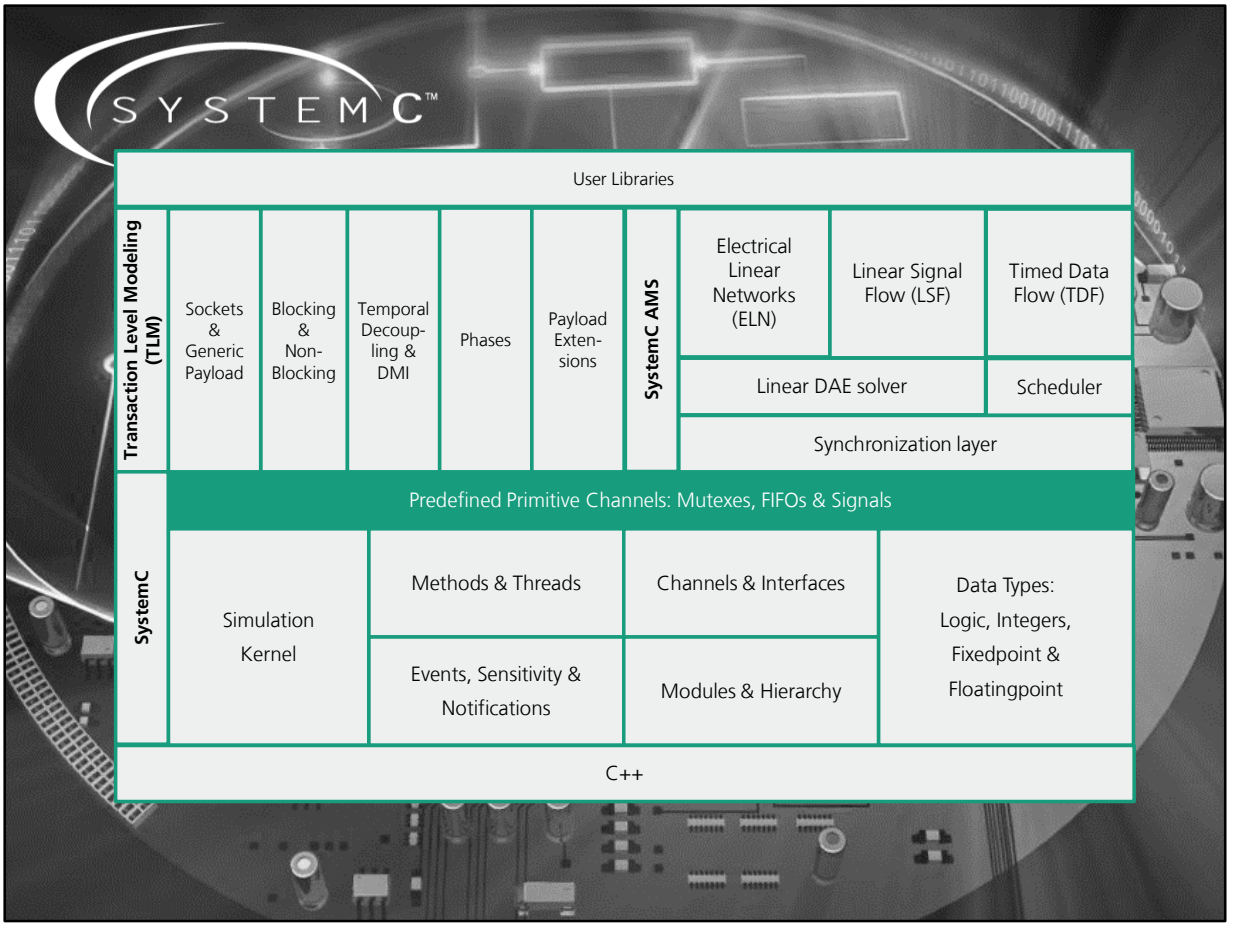
Your notes:

Note on Indirect Waits

- 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

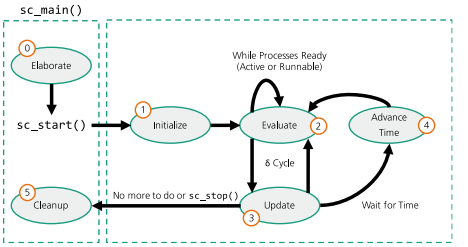
Your notes:


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++

Your notes:

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`

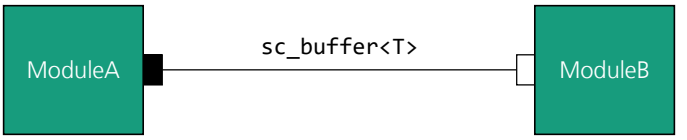


Fraunhofer IESE

Your notes:

Primitive Channels: `sc_buffer<T>`

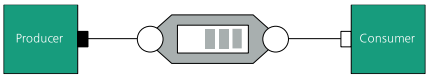
- This class is derived from `sc_signal` and has the same methods and operators
- The difference to `sc_signal` is that with `sc_buffer` an event is generated each time the `write()` method is called
- Therefore, corresponding processes sensitive to that buffer are executed.
- With `sc_signal`, an event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



Fraunhofer IESE

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


```
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?

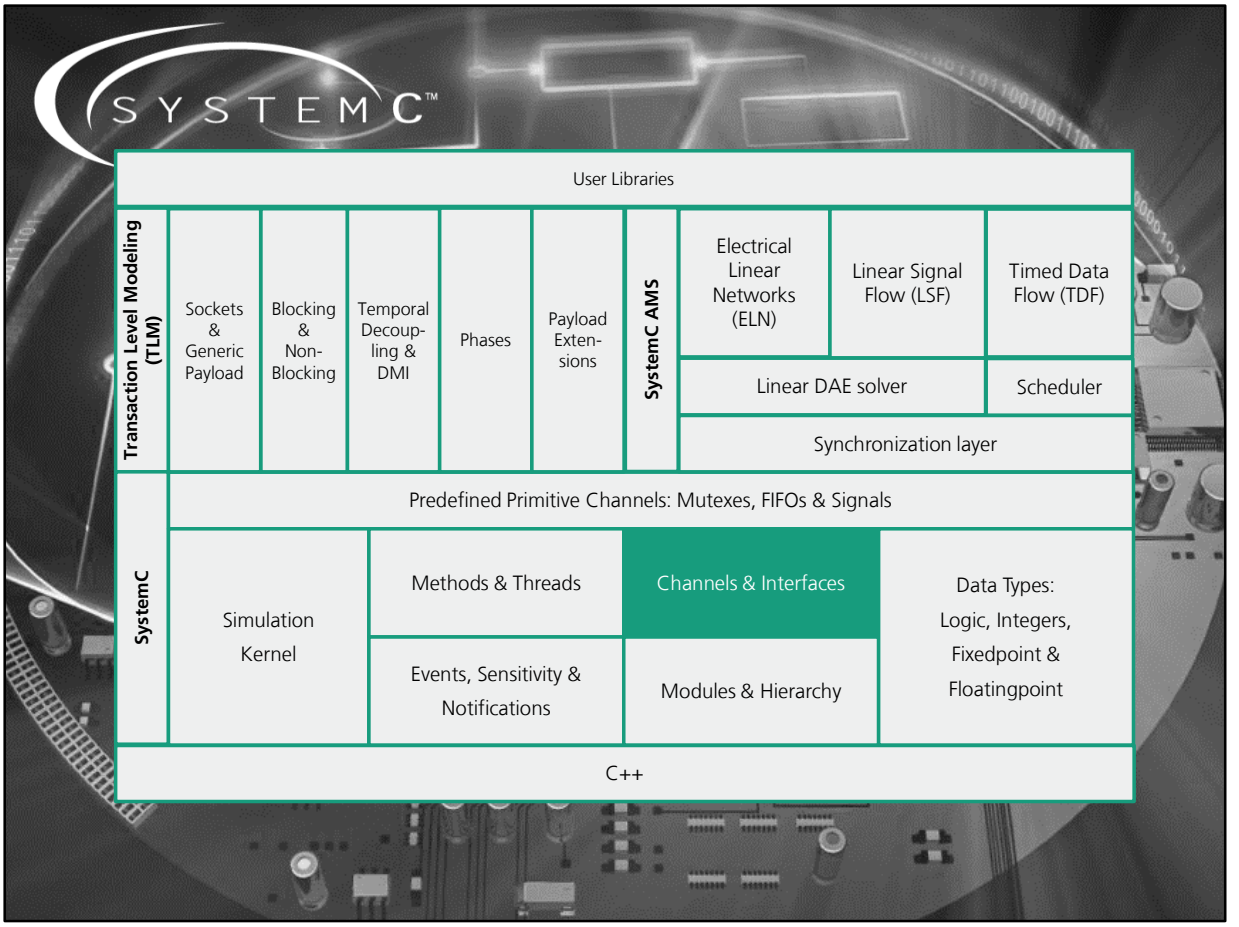
- To provide variability and interoperability in modeling
- Example 2 memory channels with same interface but different implementation:

```
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 …
}
```

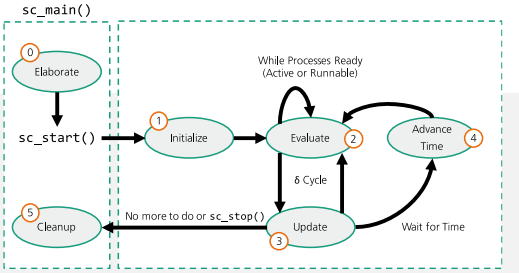

Fraunhofer IESE

Your notes:


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++

Your notes:

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;
};
```


Fraunhofer IESE

Your notes:

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;
    }
};
```

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

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

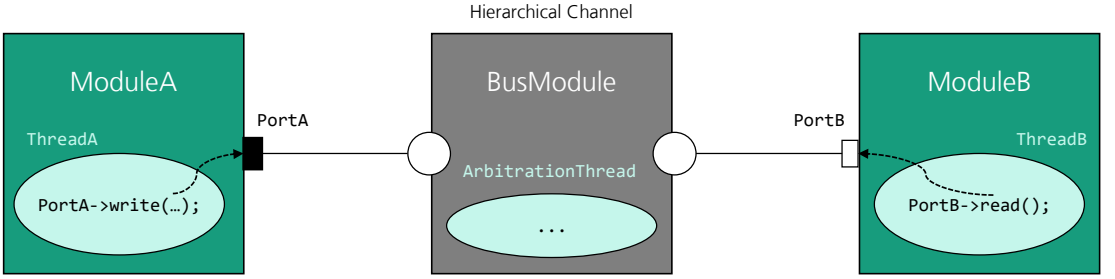
35

Fraunhofer IESE

Your notes:

Hierarchical Channels

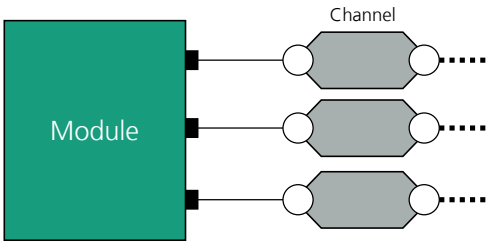
- Primitive channels are derived from `sc_prim_channel` and are "*pasive*"
- Hierarchical Channels are derived from `sc_module` and can be "*active*"
  - Hierarchical Channels use also the concept of Interfaces
  - They can have internal `SC_THREAD`s and `SC_METHOD`s
  - They can consist of other `sc_module`s,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



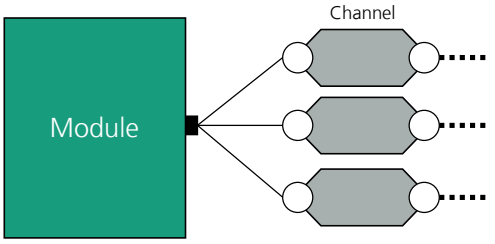
Fraunhofer IESE

Your notes:

Ports, Port Arrays and Multiports



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

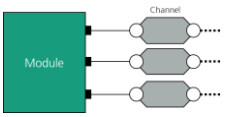


- Dynamic port creation during elaboration phase
- Using Multiports

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;
}
```

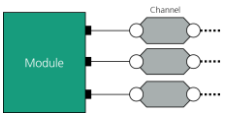
- Static port creation at compile time
- Connected channels are addressed with [ ] operator

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

Fraunhofer IESE

Your notes:

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);
        }
    }
};
```

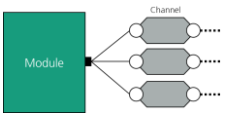
```
int sc_main(…)
{
    module<3> m("m");
    sc_fifo<int> f1, f2, f3;

    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

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!

Try code on github:

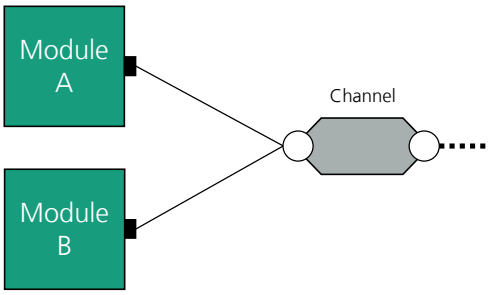
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/multiports

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

Fraunhofer IESE

Your notes:

Multiple Bindings

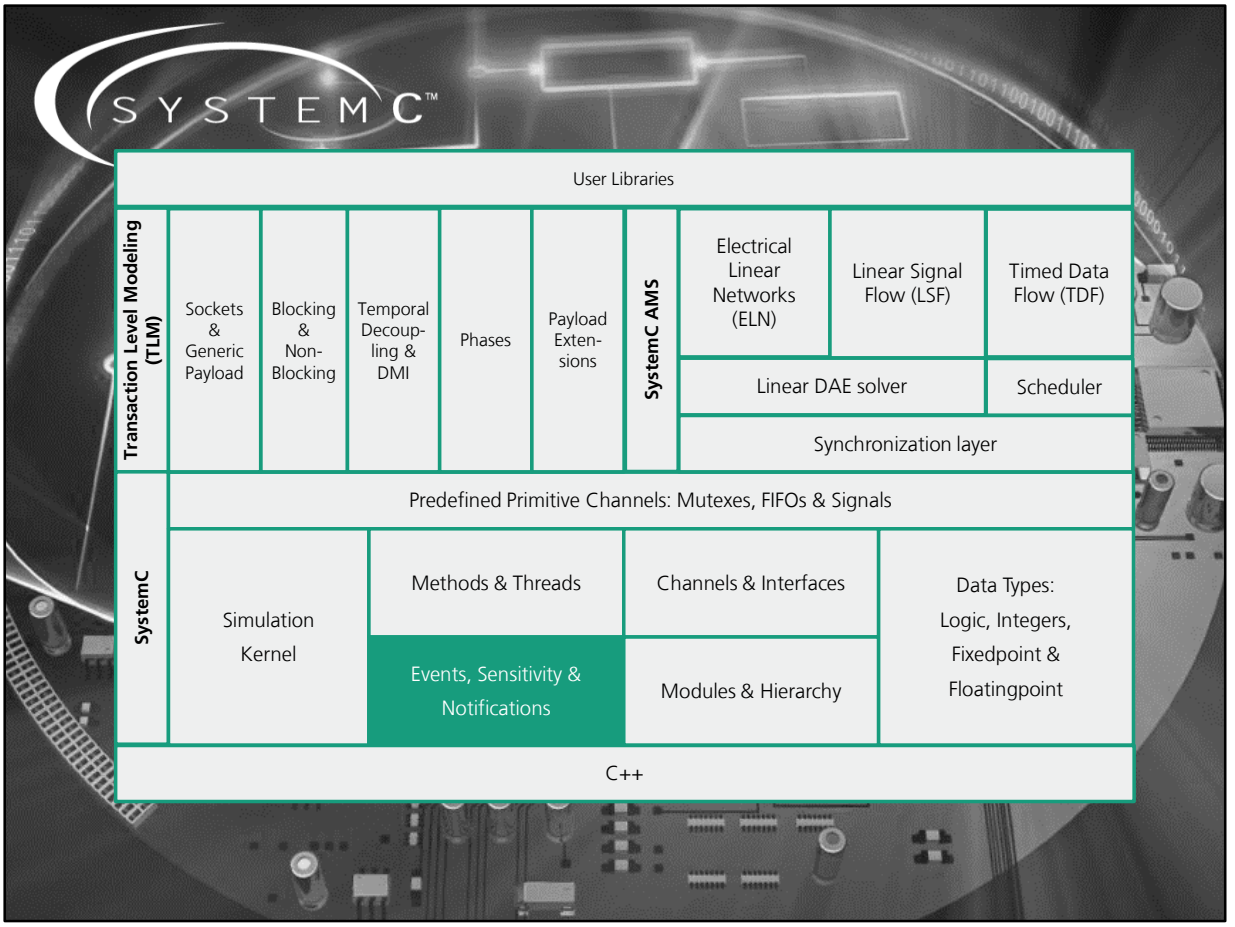



```
int sc_main(…)
{
    module a("a");
    module b("b");

    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:


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++

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;
    }
};
```

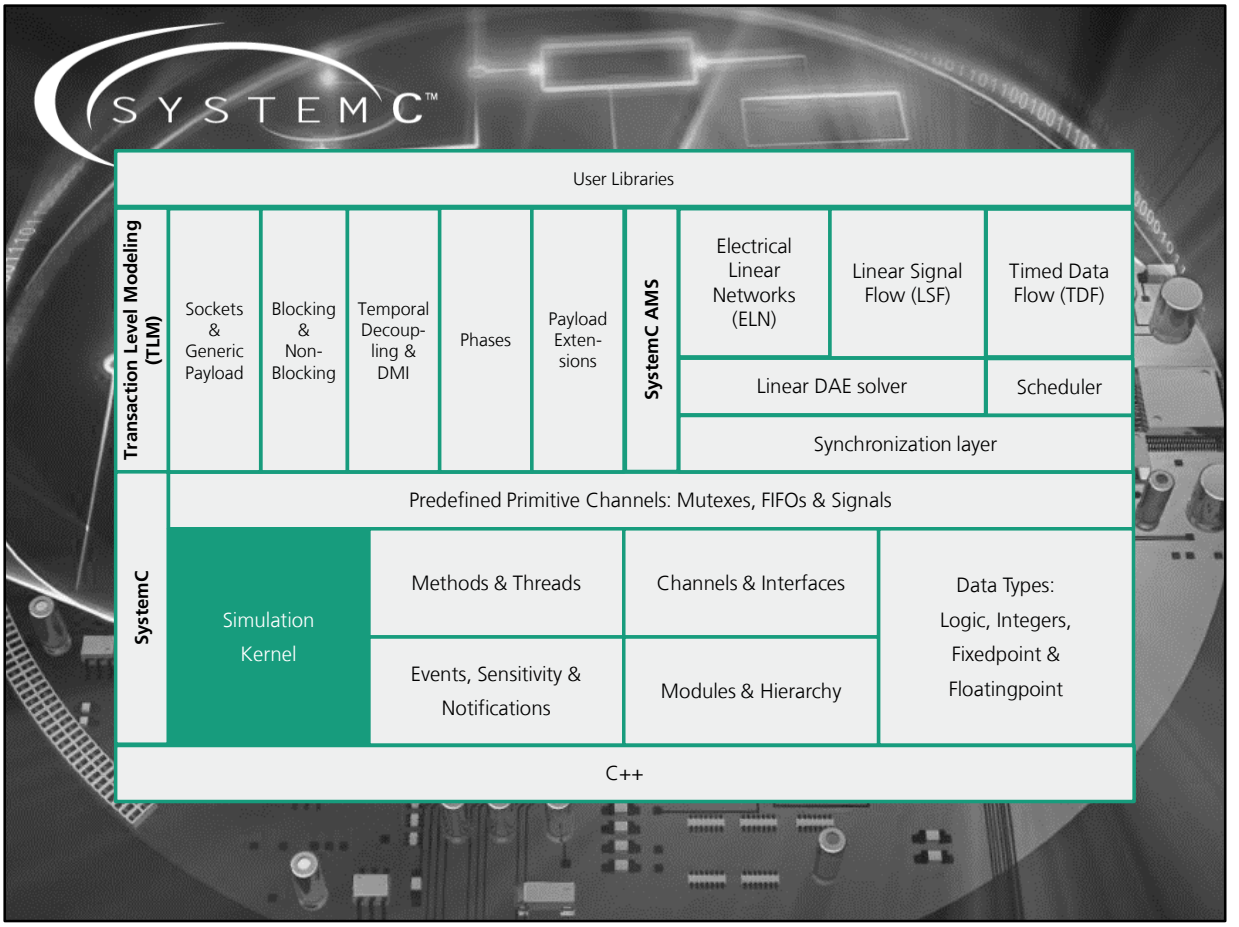
- The class `sc_event_queue` notes all notifications
- Orders events w.r.t ascending time
- Provides also interface `sc_event_queue_if` for using as a port

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:


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++

Your notes:

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!

Fraunhofer IESE

Your notes:

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

Your notes:

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:

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

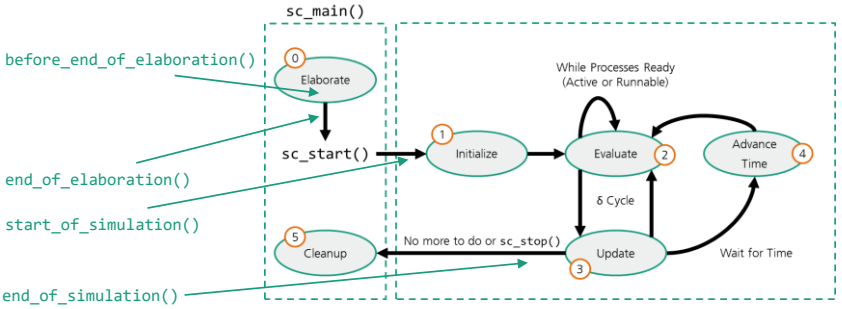
Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/reporting

Fraunhofer IESE

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:

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

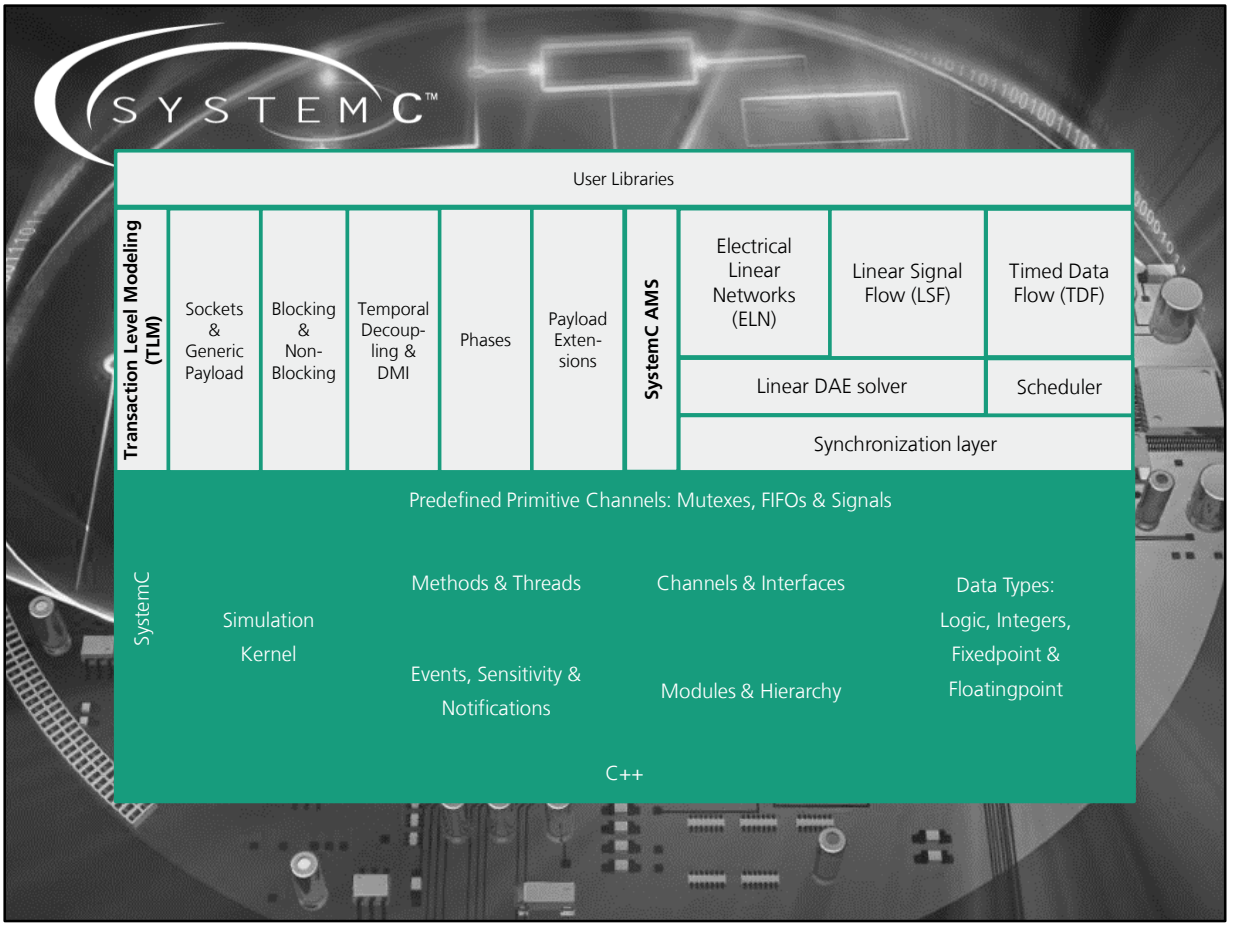
```
SC_MODULE(module){
    public:
    sc_in<bool> clk;
    sc_trace_file *tf;

    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

56

Fraunhofer IESE

Your notes:


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++

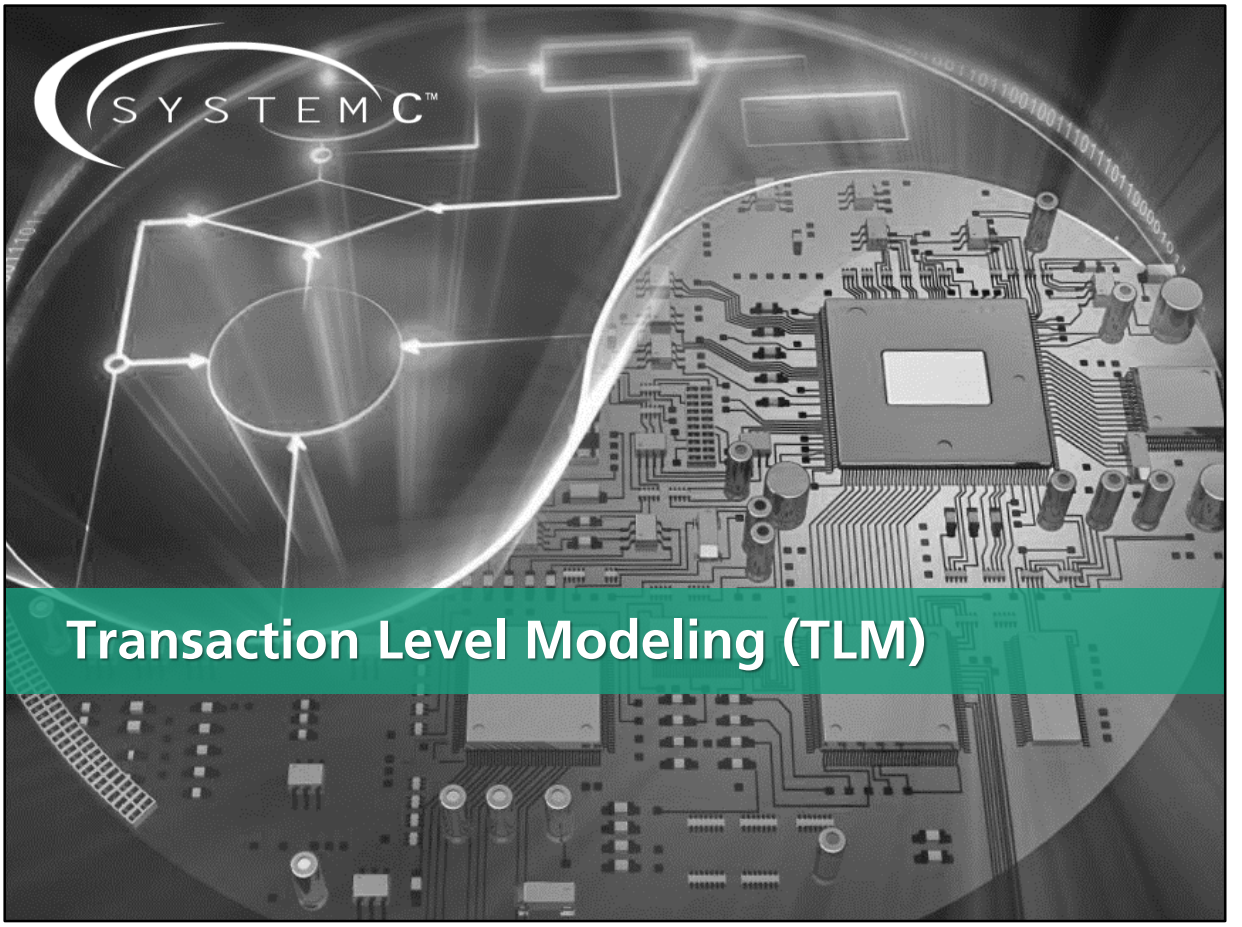
Your notes:

SystemC and Virtual Prototyping

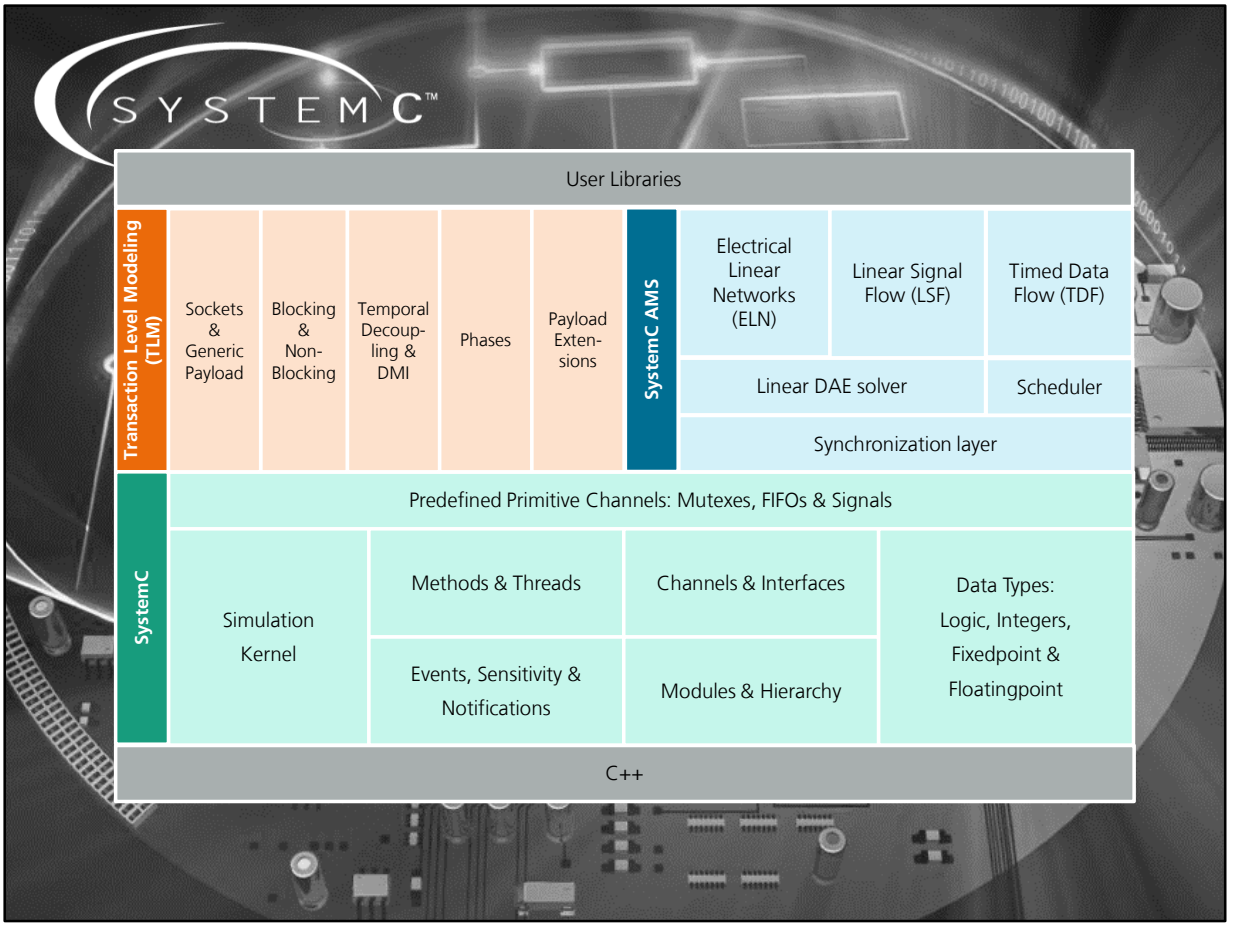
Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

Fraunhofer IESE

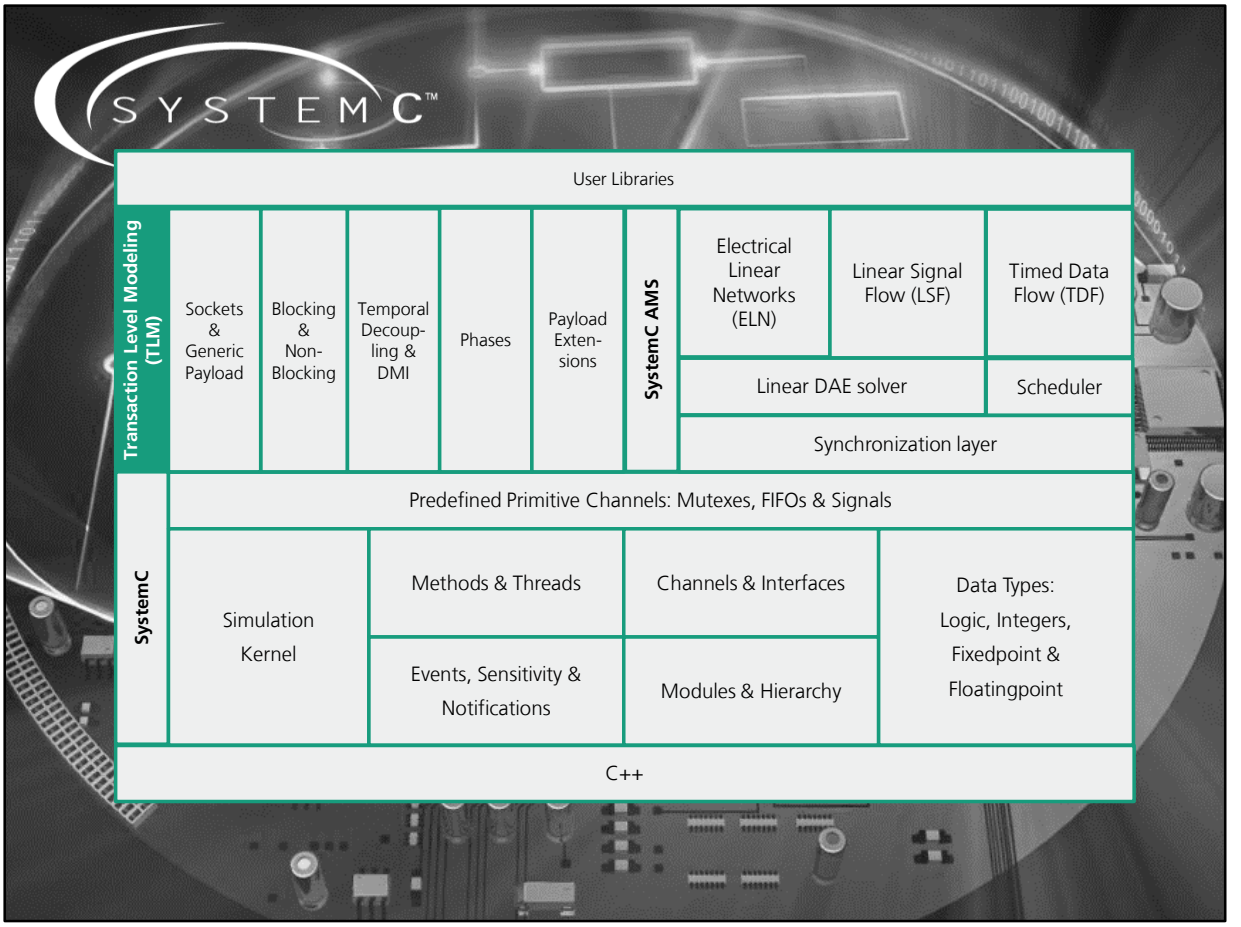
Your notes:



Your notes:


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++

Your notes:


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++

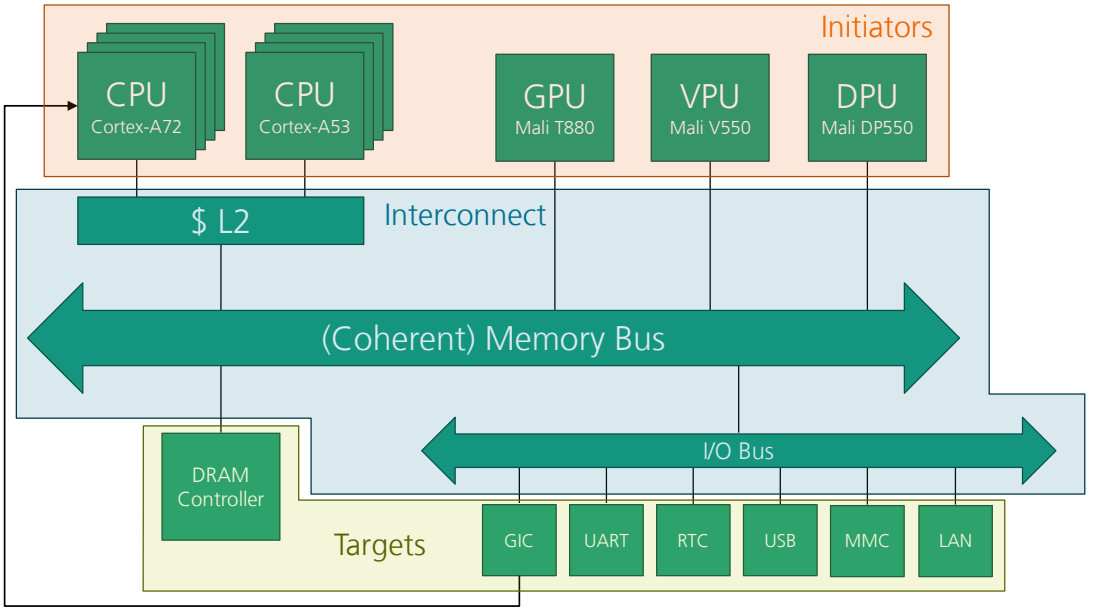
Your notes:

How to build a virtual Smartphone?

Fraunhofer IESE

Your notes:

Typical Smartphone SoC



Fraunhofer IESE

Your notes:

Recap: Accuracy vs. Speed Trade-Off

Fraunhofer IESE

Your notes:

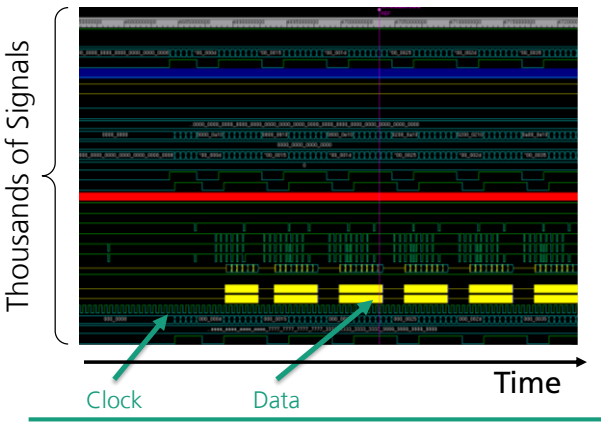
Recap: Accuracy vs. Speed Trade-Off

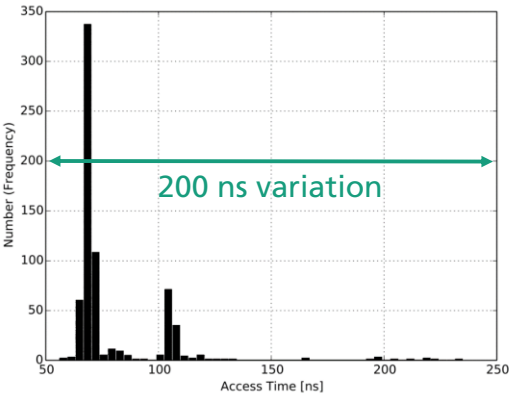
E.g. RTL Simulations:

- VHDL / Verilog / SystemC
- Very accurate
- Very very very … slow
- Inflexible

E.g. System Level Simulations:

- Fast
- Large flexibility
- Inaccurate



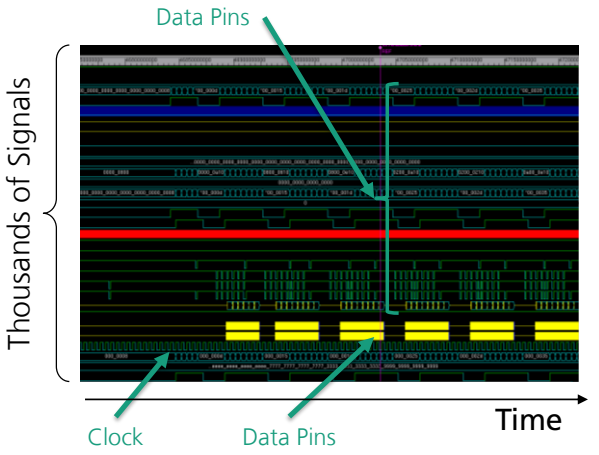


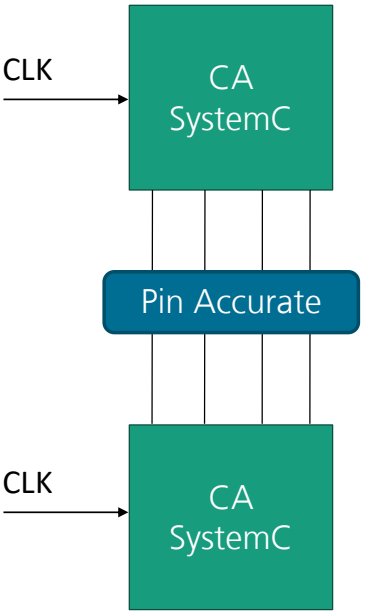
9

Fraunhofer IESE

Your notes:

Remember: Cycle Accurate Simulation





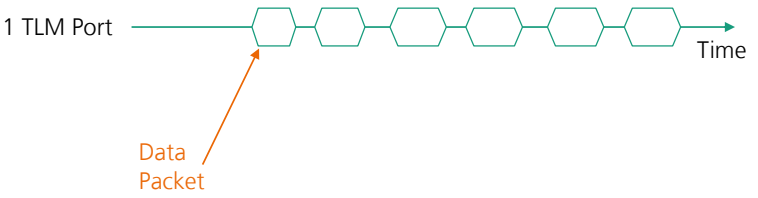
- RTL models can have thousands of pins
- Simulate all events on all pins
  - RTL bus access has ~75 events

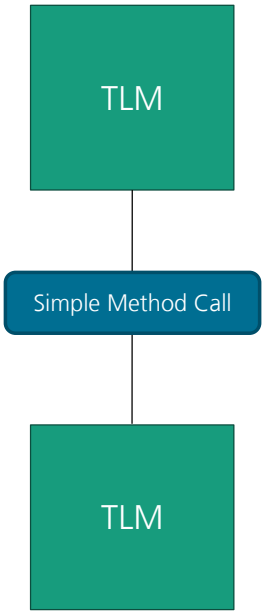
Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Your notes:

Transaction Level Modeling (TLM)





- Reduce structural accuracy by replacing signals by single function calls
  - e.g. AXI/AHB require more than 100 signals
- TLM is communication centric
  - Concentrate only on the important events
  - i.e. the Transacitons
  - TLM bus access has 1-4 events
- TLM Simulations 100-10,000 times faster than RTL

11

Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Transaction level modeling is speeding up simulation by replacing all pin-level events with a single method call. In an RTL simulation the communication requires multiple events at the level of individual bits. In a transaction-level model, complete bus cycles can be modeled with one method call. Therefore, TLM models can be 100 to 10000 times faster than their RTL counter parts. However, there is no free lunch: the increased speedup of simulation is payed with reduced accuracy of the results.

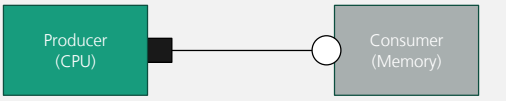
Your notes:

Transaction: A custom TLM Implementation

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

using namespace std;

enum cmd {READ, WRITE};

struct transaction {
    unsigned int data;
    unsigned int addr;
    cmd command;
};

class transactionInterface : public sc_interface {
    public:
    virtual void transport(transaction &trans) = 0;
};
```

Try code on github:

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



Consumer Module as Hierarchical Channel

Fraunhofer IESE

Your notes:

Transaction: A custom TLM Implementation

```
SC_MODULE(PRODUCER)
{
    sc_port< transactionInterface > master;

    SC_CTOR(PRODUCER)
    {
        SC_THREAD(process);
    }

    void process()
    {
        for(unsigned int i=0; i < 4; i++) {
            wait(1,SC_NS);
            transaction trans;
            trans.addr = i;
            trans.data = rand();
            trans.command = cmd::WRITE;
            master->transport(trans);
        }

        for(unsigned int i=0; i < 4; i++) {
            wait(1,SC_NS);
            transaction trans;
            trans.addr = i;
            trans.data = 0;
            trans.command = cmd::READ;
            master->transport(trans);
            cout << trans.data << endl;
        }
    }
};
```

Write

Read

```
class CONSUMER : public sc_module,
                 public transactionInterface
{
    private:
    unsigned int memory[1024];

    public:
    SC_CTOR(CONSUMER) {
        for(unsigned int i=0; i < 1024; i++) {
            memory[i] = 0; // Initialize memory
        }
    }

    void transport(transaction &trans) {
        if(trans.command == cmd::WRITE) {
            memory[trans.addr] = trans.data;
        }
        else /* cmd::READ */ {
            trans.data = memory[trans.addr];
        }
    }
};

int sc_main(…) {
    PRODUCER cpu("cpu");
    CONSUMER mem("memory");
    cpu.master.bind(mem);
    sc_start();
    return 0;
}
```

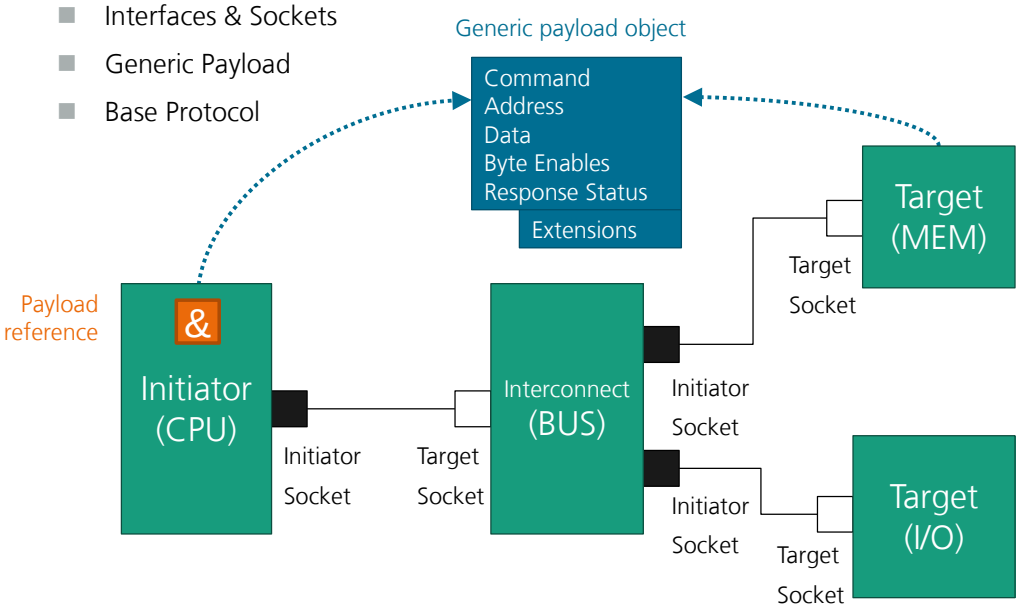
The Produce is active, the consumer is passive

Fraunhofer IESE

Your notes:

TLM Basic Concept

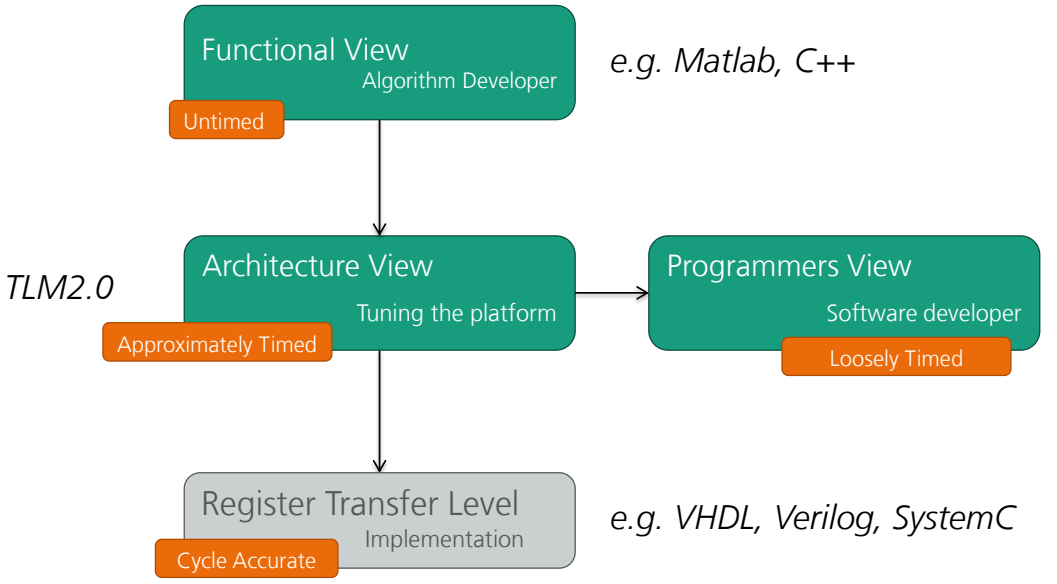
- TLM2.0 Interoperability Layer:
  - Interfaces & Sockets
  - Generic Payload
  - Base Protocol



Fraunhofer IESE

Your notes:

TLM Use Cases / Timing Accuracy

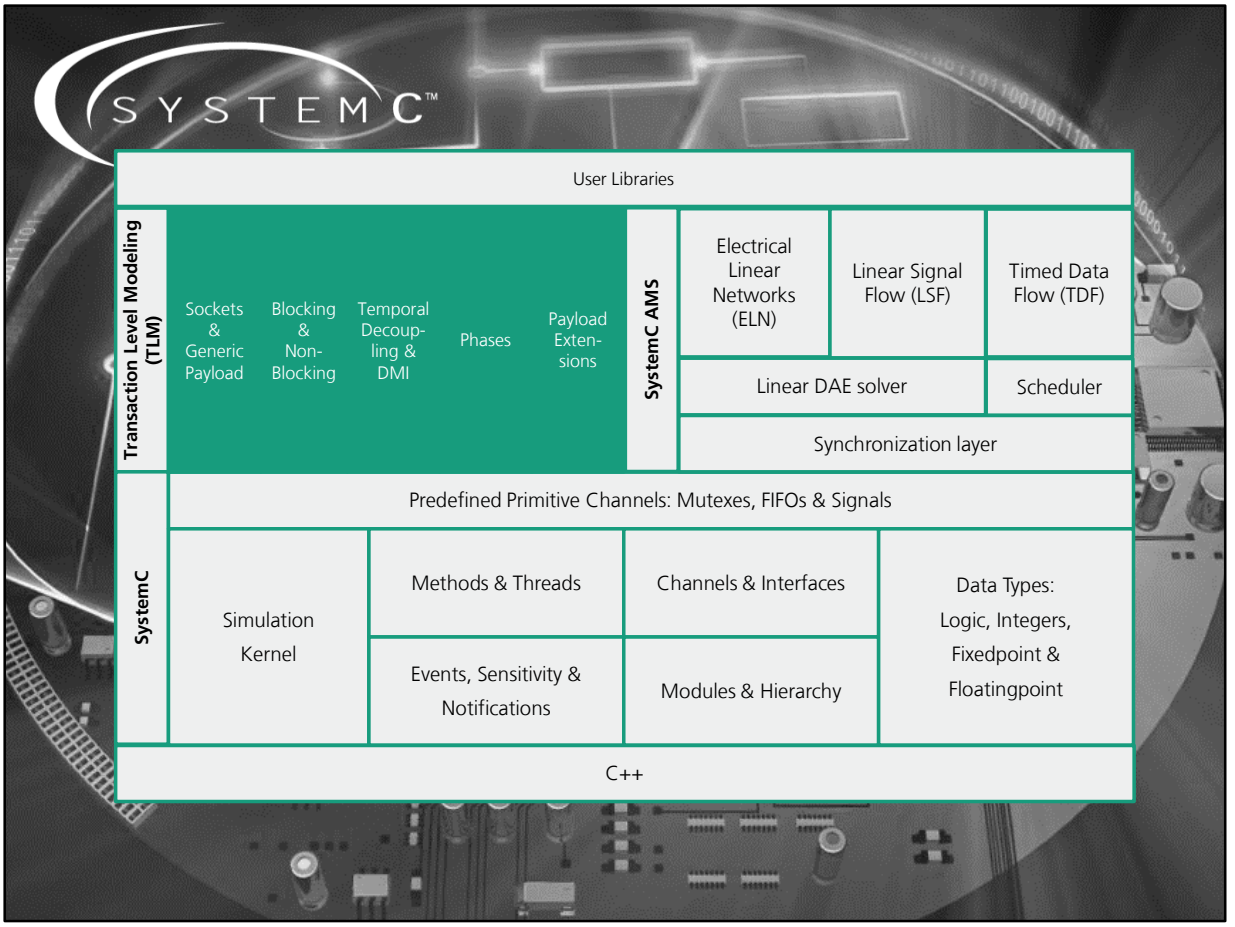


15

Source: Doulos Ldt. www.doulos.com

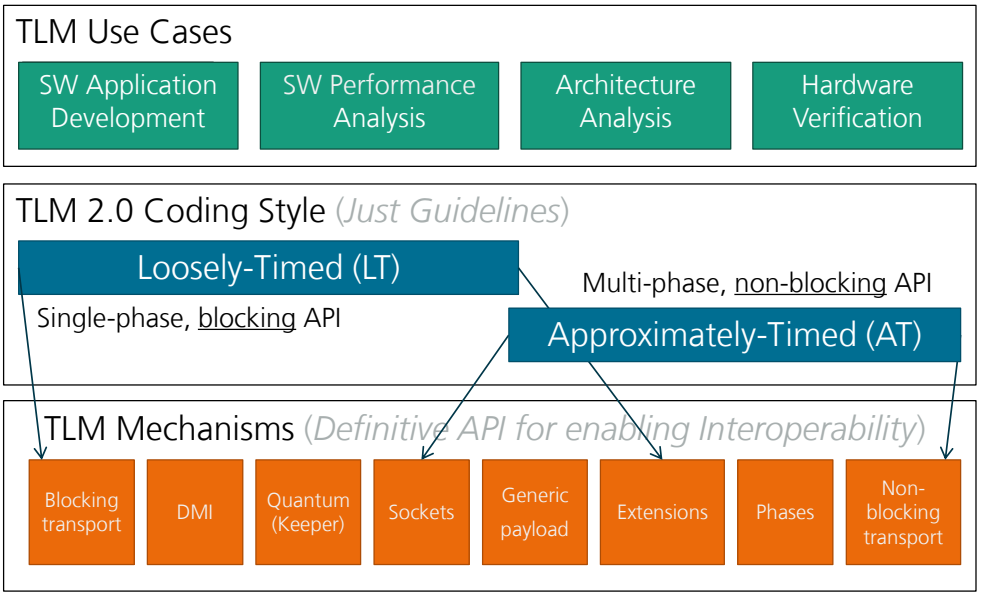
Fraunhofer IESE

Your notes:


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++

Your notes:

TLM Coding Styles and Mechanisms

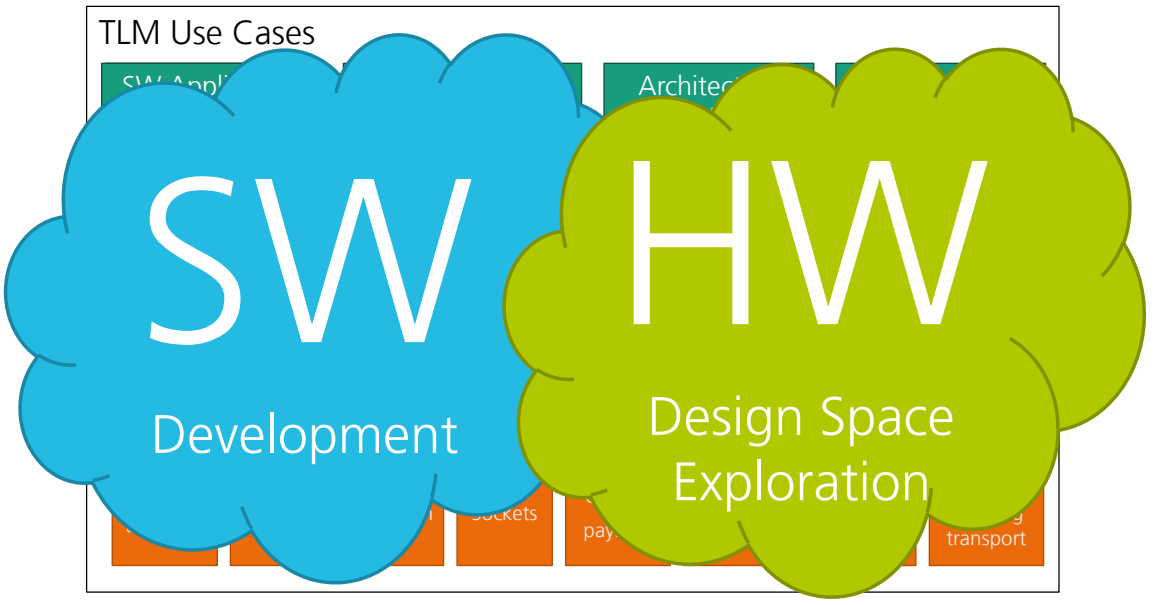


Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Your notes:

TLM Coding Styles and Mechanisms



19

Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Your notes:

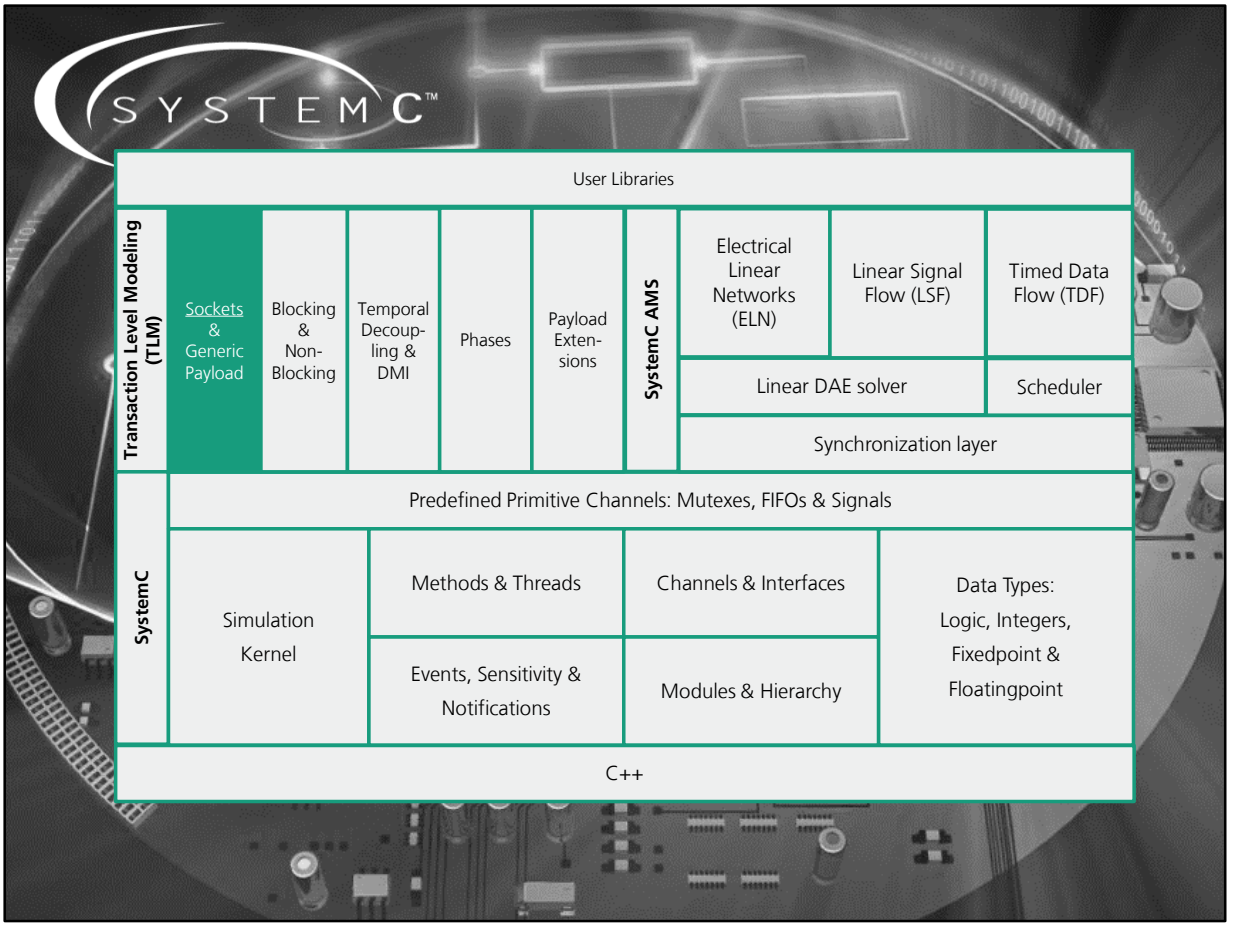
Coding Styles in TLM

- **Loosely-Timed (LT):**
  - As fast as possible
  - Sufficient timing detail to boot OS and run multicore systems and to develop SW or drivers
  - Processes can run ahead of simulation time (temporal decupling)
  - Each transaction completes in one blocking function call
  - Usage of Direct Memory Interface (DMI) e.g. for boot process

- **Approximately-Timed (AT):**
  - Accurate enough for performance modelling
  - Sufficient for architectural HW design space exploration
  - Processes run in lockstep with simulation time
  - Each transaction has usually 4 timing points i.e. 4 function calls (extensible if required, also less possible); non-blocking behavior
  - More detailed than LT and therefore also slower than LT

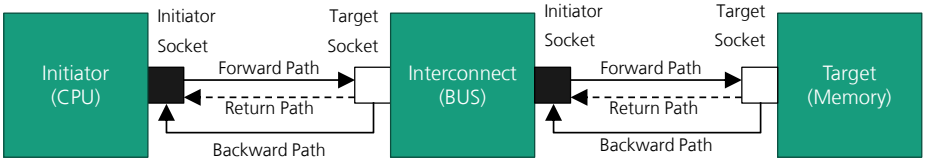
Fraunhofer IESE

Your notes:


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++

Your notes:

Initiators, Targets and Interconnect

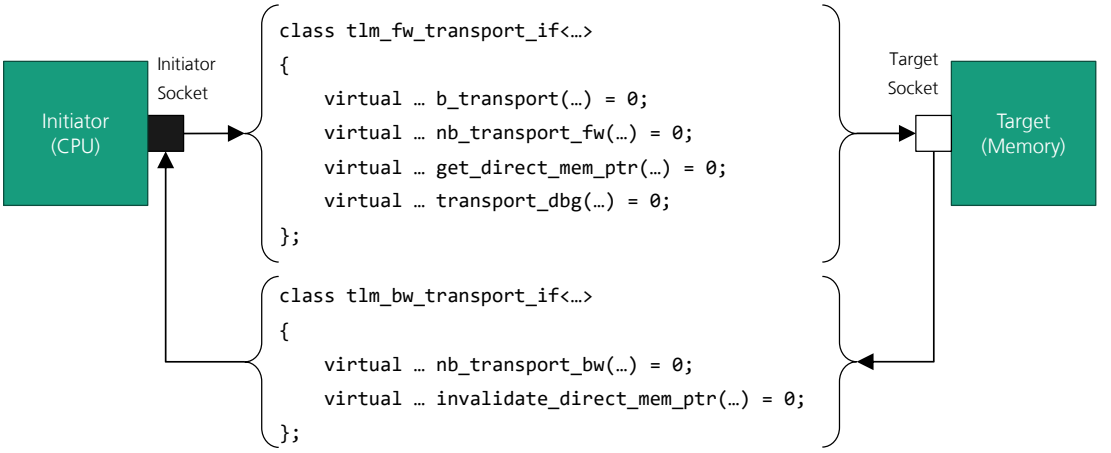


- TLM components are divided into Initiators, Targets and Interconnect components:
  - A initiator initiates (construct and send) new transactions
  - A target is a module that acts as the end point for a transaction. It executes requests from an initiator and send responses
  - An interconnect component forwards and routes transaction objects between initiators and targets
- Transactions are sent through initiator sockets (■) and received through target sockets (□)
- References to the object are passed along the forward and backward paths:
  - LT uses Forward and Return Path
  - AT uses Forward, Backward and Return Path

Fraunhofer IESE

Your notes:

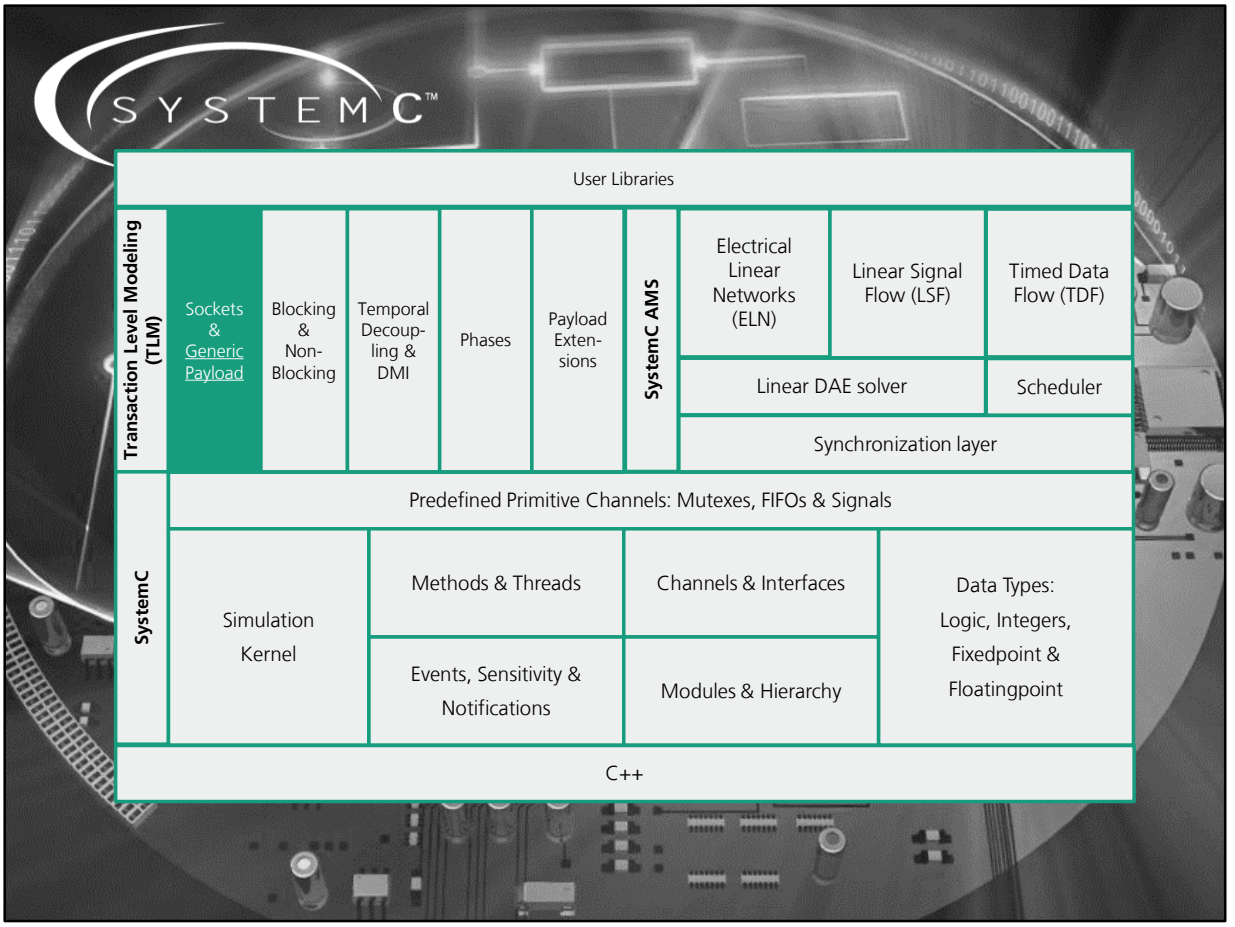
Initiator and Target Sockets



- Target port must implement `tlm_fw_transport_if` methods
- Initiator must implement `tlm_bw_transport_if` methods
- `b_transport` and `mem_ptr` functions are used for LT modeling
- `nb_transport` functions are used for AT modeling

Fraunhofer IESE

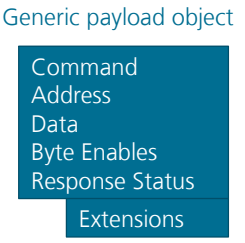
Your notes:


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++

Your notes:

Generic Payload

- The generic payload is designed to include typical attributes of memory mapped busses (e.g. AXI, AHB, etc.)
- It can supports
  - READ and WRITE transactions
  - Byte enables
  - Single word transfer
  - Burst transfer
  - Streaming transfer
- Extensions can be used to carry further metadata or to model more complex bus and NoC protocols and maintain 100% compatibility because they are ignorable
- Very efficient implementation for simulation speed



Fraunhofer IESE

Your notes:

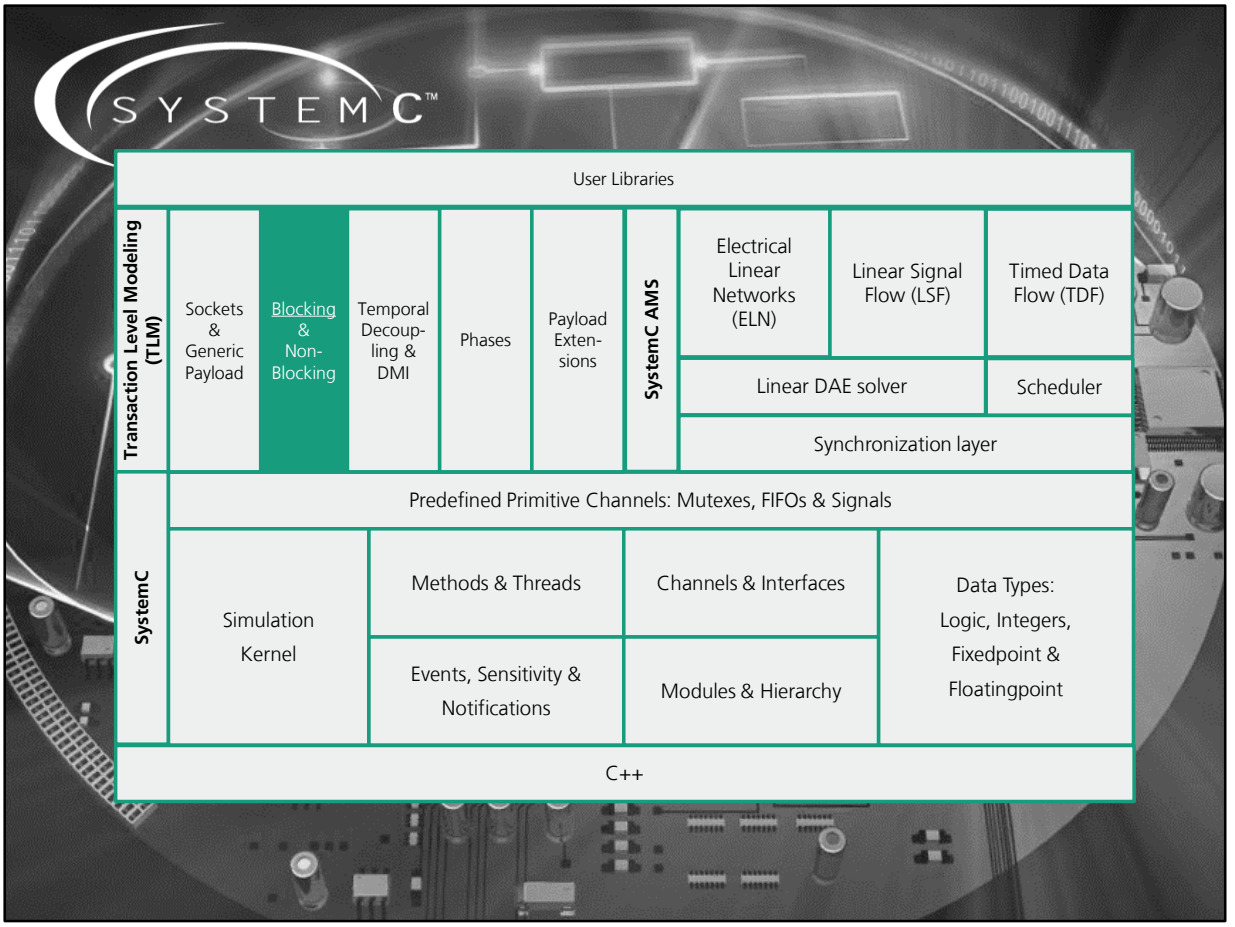
Generic Payload Attributes

Attribute	Type	Modifiable?
Command	`tlm_command`	No
Address	`uint64_t`	Interconnect Only
Data Pointer	`unsinged char *`	No (array yes)
Data Length	`unsigned int`	No
Byte Enable Pointer	`unsigned char *`	No
Byte Enable Length	`unsinged int`	No
Streaming Width	`unsigned int`	No
DMI Hint	`bool`	Yes
Response Status	`tlm_response_status`	Target Only
Extensions	`(tlm_extension_base*)[]`	Yes

- Initiator should initialize attributes before sending the transaction object
- Set-Methods like `set_address()` etc. are provided
- The majority of the attributes must not be changed by Interconnects & Targets

Fraunhofer IESE

Your notes:


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++

Your notes:

Building a Blocking (LT) Initiator

SW

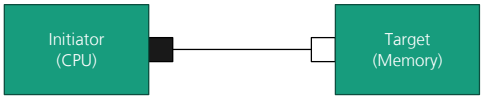
```
class Initiator: sc_module, tlm::tlm_bw_transport_if<> {
    public:
    tlm::tlm_initiator_socket<> iSocket;
    SC_CTOR(Initiator) : iSocket("iSocket") {
        iSocket.bind(*this);
        SC_THREAD(process);
    }

    void process() {
        for (int i = 0; i < 4; i++) {
            tlm::tlm_generic_payload trans;
            unsigned char data = rand();
            trans.set_address(i);
            trans.set_data_length(1);
            trans.set_command(tlm::TLM_WRITE_COMMAND);
            trans.set_data_ptr(&data);
            sc_time delay = sc_time(0, SC_NS);
            iSocket->b_transport(trans, delay);
            wait(delay);
        }
        for (int i = 0; i < 4; i++) {
            tlm::tlm_generic_payload trans;
            unsigned char data;
            trans.set_address(i);
            trans.set_data_length(1);
            trans.set_command(tlm::TLM_READ_COMMAND);
            trans.set_data_ptr(&data);
            sc_time delay = sc_time(0, SC_NS);
            iSocket->b_transport(trans, delay);
            wait(delay);
        }
    }
```

Write to memory

Read from memory

```
    // Dummy method:
    void invalidate_direct_mem_ptr(
        sc_dt::uint64 start_range,
        sc_dt::uint64 end_range)
    {}

    // Dummy method:
    tlm::tlm_sync_enum nb_transport_bw(
            tlm::tlm_generic_payload& trans,
            tlm::tlm_phase& phase,
            sc_time& delay)
    {
        return tlm::TLM_ACCEPTED;
    }
};
```

Must be implemented



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

Fraunhofer IESE

Your notes:

Building a Blocking (LT) Target


```
class Target:sc_module, tlm::tlm_fw_transport_if<> {
    private:
    unsigned char mem[1024];

    public:
    tlm::tlm_target_socket<> tSocket;

    SC_CTOR(Target) : tSocket("tSocket") {
        tSocket.bind(*this);
    }

    void b_transport(tlm::tlm_generic_payload &trans,
                     sc_time &delay)
    {
       if(trans.get_address() >= 1024){
         SC_REPORT_FATAL(this->name(),"Out of Range");
       }

       if(trans.get_command() == tlm::TLM_WRITE_COMMAND)
       {
         memcpy(mem+trans.get_address(),  // destination
                trans.get_data_ptr(),     // source
                trans.get_data_length()); // size
       } else {
         memcpy(trans.get_data_ptr(),     // destination
                mem+trans.get_address(),  // source
                trans.get_data_length()); // size
       }
       delay = delay + sc_time(40, SC_NS);
    }
    ...
```

```
    // Dummy method
    virtual tlm::tlm_sync_enum nb_transport_fw(
            tlm::tlm_generic_payload& trans,
            tlm::tlm_phase& phase,
            sc_time& delay )
    {
       return tlm::TLM_ACCEPTED;
    }

    // Dummy method
    bool get_direct_mem_ptr(
         tlm::tlm_generic_payload& trans,
         tlm::tlm_dmi& dmi_data)
    {
        return false;
    }

    // Dummy method
    unsigned int transport_dbg(
        tlm::tlm_generic_payload& trans)
    {
        return 0;
    }
};
```

Must be implemented

Fraunhofer IESE

Your notes:

Binding Target and Initiator


```
int sc_main (…)
{
    Initiator * cpu = new Initiator("cpu");
    Target * memory = new Target("memory");

    cpu->iSocket.bind(memory->tSocket);

    sc_start();
    return 0;
}
```

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

Summary:

- Initiator and target ports are derived from `sc_port` and `sc_export`
- `b_transport` uses call by reference to transfer the transaction object (from `tlm_generic_payload` class)
- The key idea of timing annotation is that the recipient is obliged to behave as if it had received the transaction at time `sc_time_stamp() + delay`
- All virtual methods must be implemented
- Transaction objects should be reused to avoid always new allocations

Fraunhofer IESE

Your notes:

Error Handling for b_transport

enum tlm_response_status	Meaning
`TLM_OK_RESPONSE`	Successful transmission
`TLM_INCOMPLETE_RESPONSE`	Transaction not delivered to the target (default)
`TLM_ADDRESS_ERROR_RESPONSE`	Unable to work with given address
`TLM_COMMAND_ERROR_RESPONSE`	Unable to execute command (e.g write to ROM)
`TLM_BURST_ERROR_RESPONSE`	Unable to work with given datalength
`TLM_BYTE_ENABLE_ERROR_RESPONSE`	Unabel to work with byte enable
`TLM_GENERIC_ERROR_RESPONSE`	Any other error

- Initiator should set response status to `TLM_INCLOMPLETE_RESPONSE` (default)
- Targets modify the response status
- Initiator checks status of transaction when it is completed (e.g. after `b_transport`)

Fraunhofer IESE

Your notes:

Error Handling for b_transport


```
class exampleInitiator: sc_module,
tlm::tlm_bw_transport_if<>
{
   ...
   private:
   void process()
   {
      ...
      iSocket->b_transport(trans, delay);
      if(trans.is_response_error())
      {
         SC_REPORT_FATAL(name(), "Error")
      }
   }
   ...
};

class exampleTarget : sc_module,
tlm::tlm_fw_transport_if<>
{
   ...
   unsigned char mem[512];

   public:
   tlm::tlm_target_socket<> tSocket;

   SC_CTOR(exampleTarget) : tSocket("tSocket")
   {
      tSocket.bind(*this);
   }
```

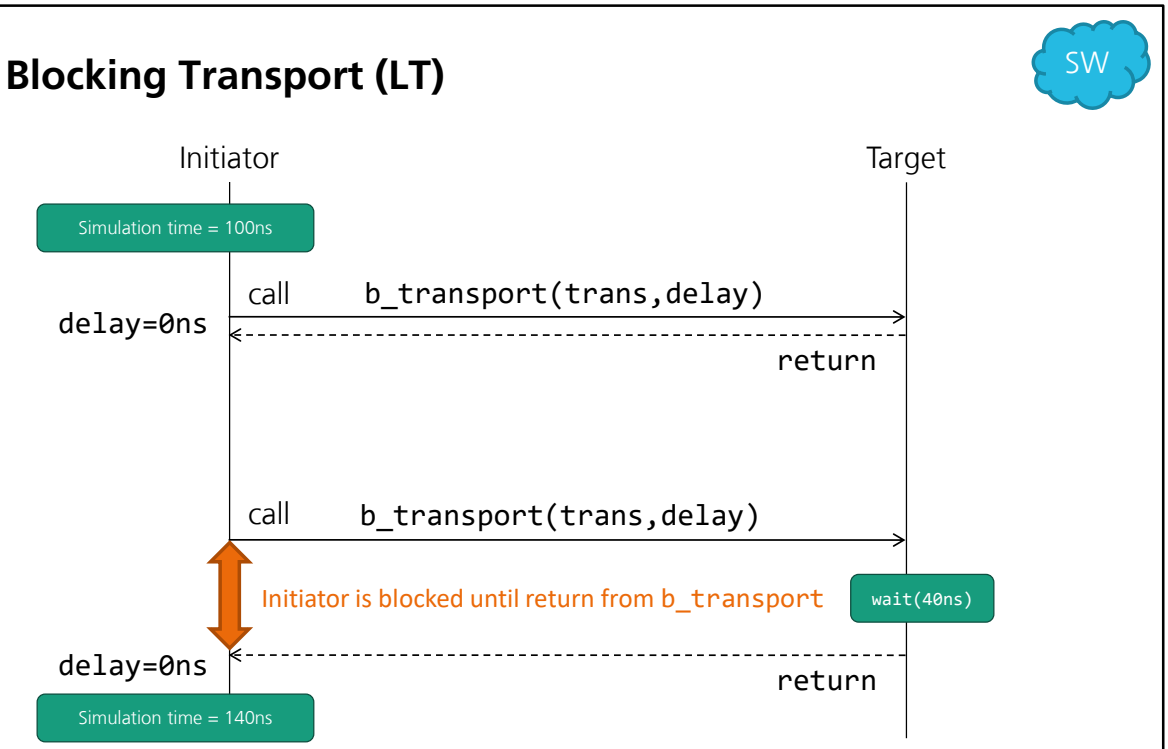
Detailed check can be done with `trans.get_response_status()`

```
   void b_transport(... &trans, ... &delay)
   {
      if (trans.get_address() >= 512) {
          trans.set_response_status(
              tlm::TLM_ADDRESS_ERROR_RESPONSE );
          return;
      }
      if (trans.get_data_length() != 4) {
          trans.set_response_status(
              tlm::TLM_BURST_ERROR_RESPONSE );
           return;
      }
      if (byt) {
          trans.set_response_status(
              tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
          return;
      }

      if(trans.get_command() == tlm::TLM_WRITE_COMMAND){
         memcpy(...);
      }
      else {
         memcpy(...);
      }

      delay = delay + sc_time(40, SC_NS);

      trans.set_response_status( tlm::TLM_OK_RESPONSE );
   }
};
```


Fraunhofer IESE

Your notes:

Blocking Transport (LT)



Calling `wait()` results in a context switch ! → bad for simulation speed

36

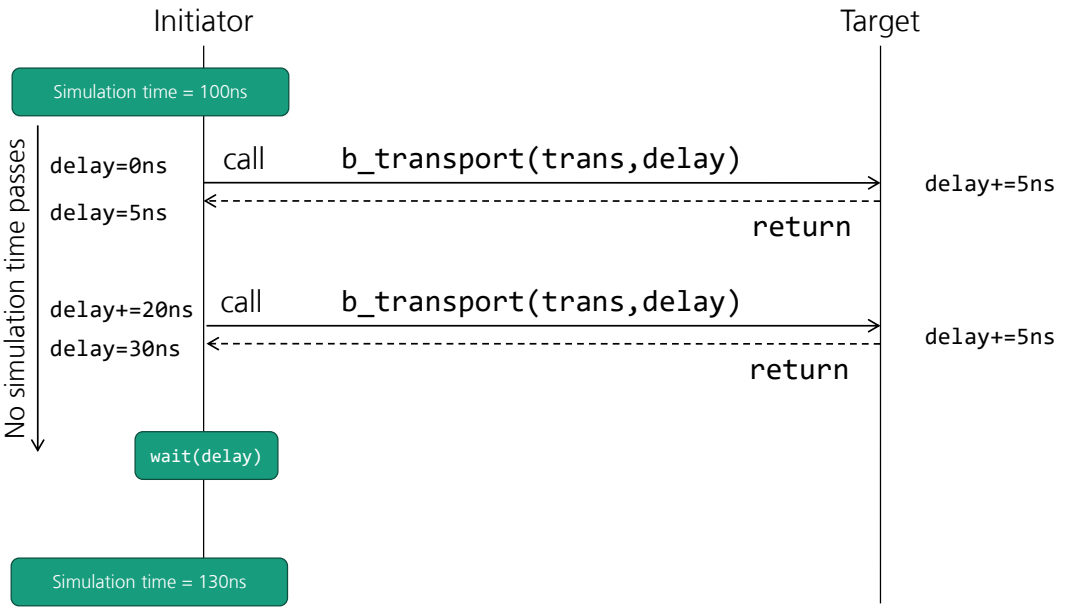
Fraunhofer IESE

An initiator, and only an initiator, may call `b_transport`. The target can return immediately, or can suspend (by calling `wait`) and return later. However, calling `wait` statements in targets should be avoided because a wait statement will result in a context switch of the simulator, which decreases the simulation speed.

The same transaction object could be reused from one call to the next, but the two calls would count as separate transactions.

Your notes:

Blocking Transport (LT)

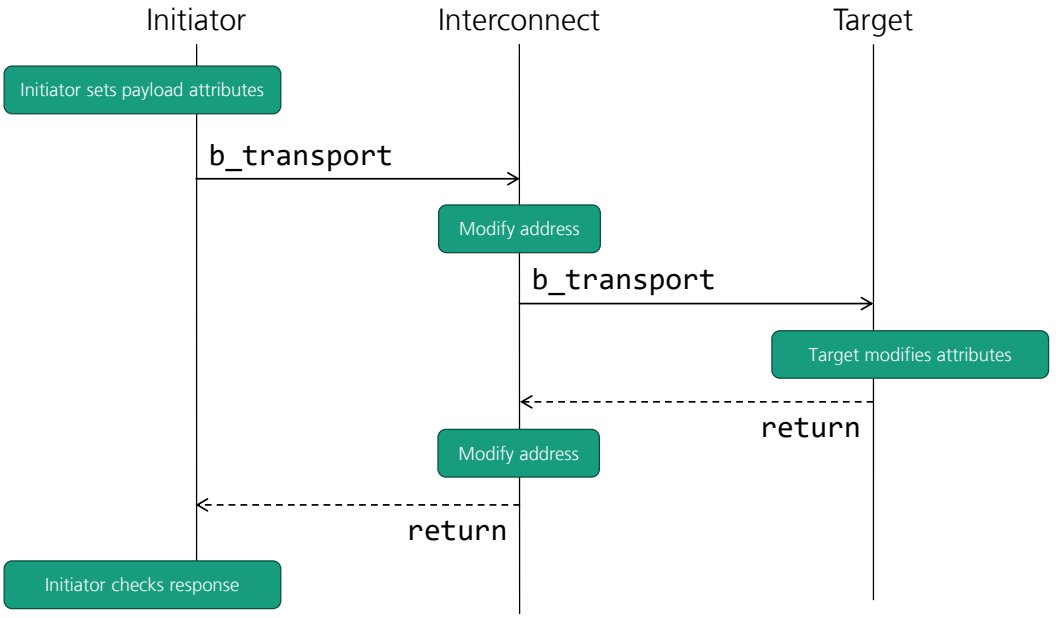


Initiator should use a local time variable and should call `wait()`! → Less context switches

Fraunhofer IESE

Your notes:

Chaining b_transport Calls



38

Fraunhofer IESE

Your notes:

Building an Interconnect Component

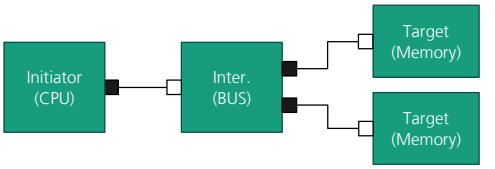
SW

```
class Interconnect : sc_module,
                     tlm::tlm_bw_transport_if<>,
                     tlm::tlm_fw_transport_if<>
{
    public:
    tlm::tlm_initiator_socket<> iSocket[2];
    tlm::tlm_target_socket<> tSocket;

    SC_CTOR(exampleInterconnect)
    {
      tSocket.bind(*this);
      iSocket[0].bind(*this);
      iSocket[1].bind(*this);
    }

    void b_transport(
        tlm::tlm_generic_payload &trans,
        sc_time &delay)
    {
      delay = delay + sc_time(40, SC_NS);

      if(trans.get_address() < 512) {
        iSocket[0]->b_transport(trans, delay);
      }
      else {
        trans.set_address(trans.get_address() - 512);
        iSocket[1]->b_transport(trans, delay);
      }
    }
    … // Dummy Methods
};
```

Annotate Bus Time

Modify address

```
int sc_main (int __attribute__((unused)) sc_argc,
             char __attribute__((unused))
*sc_argv[])
{

    Initiator * cpu    = new Initiator("cpu");
    Target * memory1   = new Target("memory1");
    Target * memory2   = new Target("memory2");
    Interconnect * bus = new Interconnect("bus");

    cpu->iSocket.bind(bus->tSocket);
    bus->iSocket[0].bind(memory1->tSocket);
    bus->iSocket[1].bind(memory2->tSocket);

    sc_start();

    return 0;
}
```



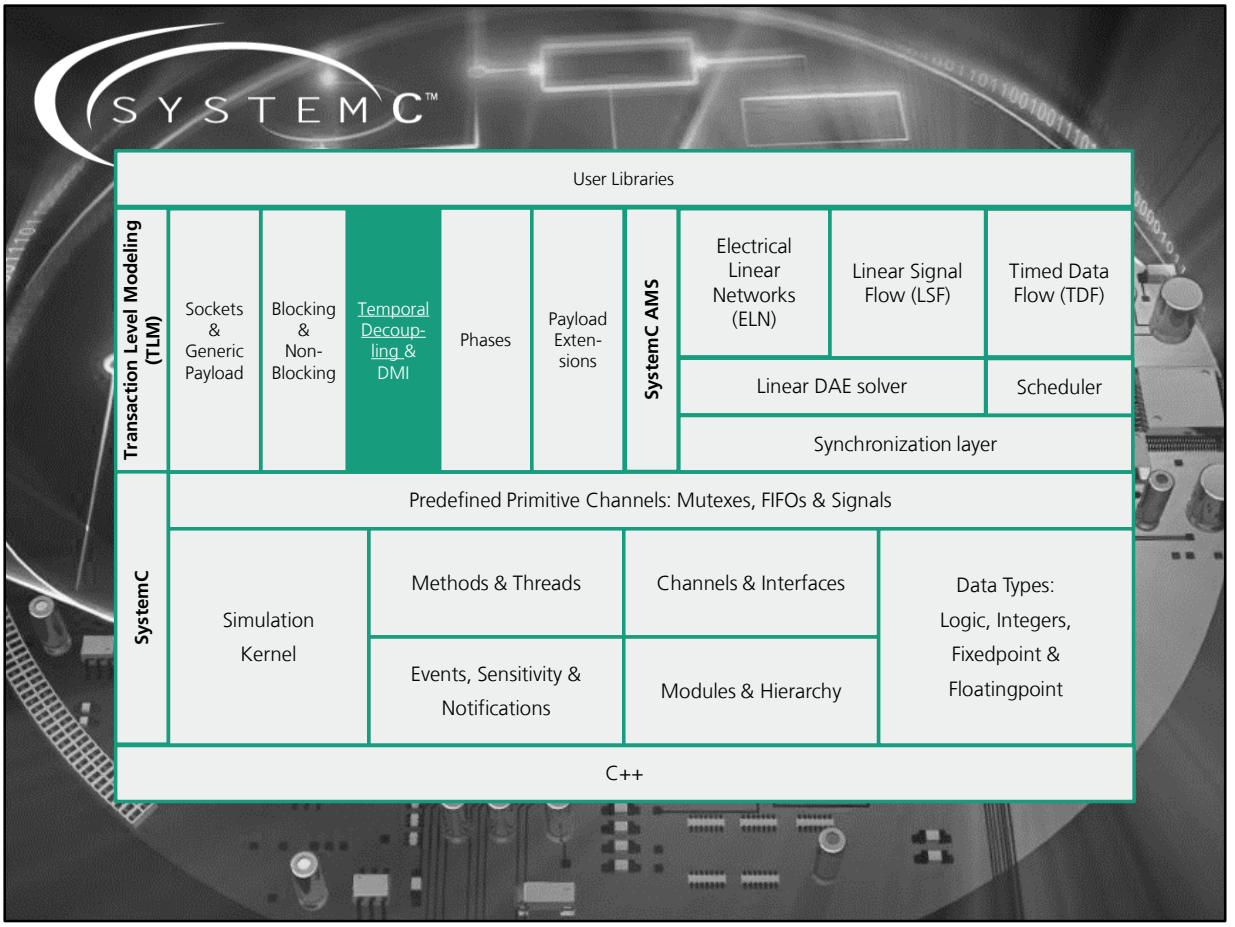
Try code on github:

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

39

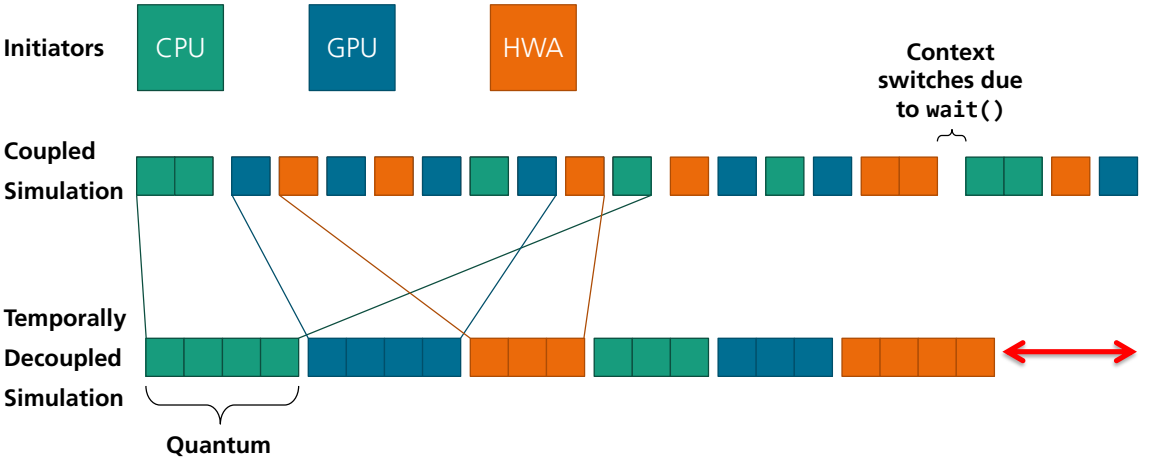
Fraunhofer IESE

Your notes:


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++

Your notes:

Temporal Decoupling

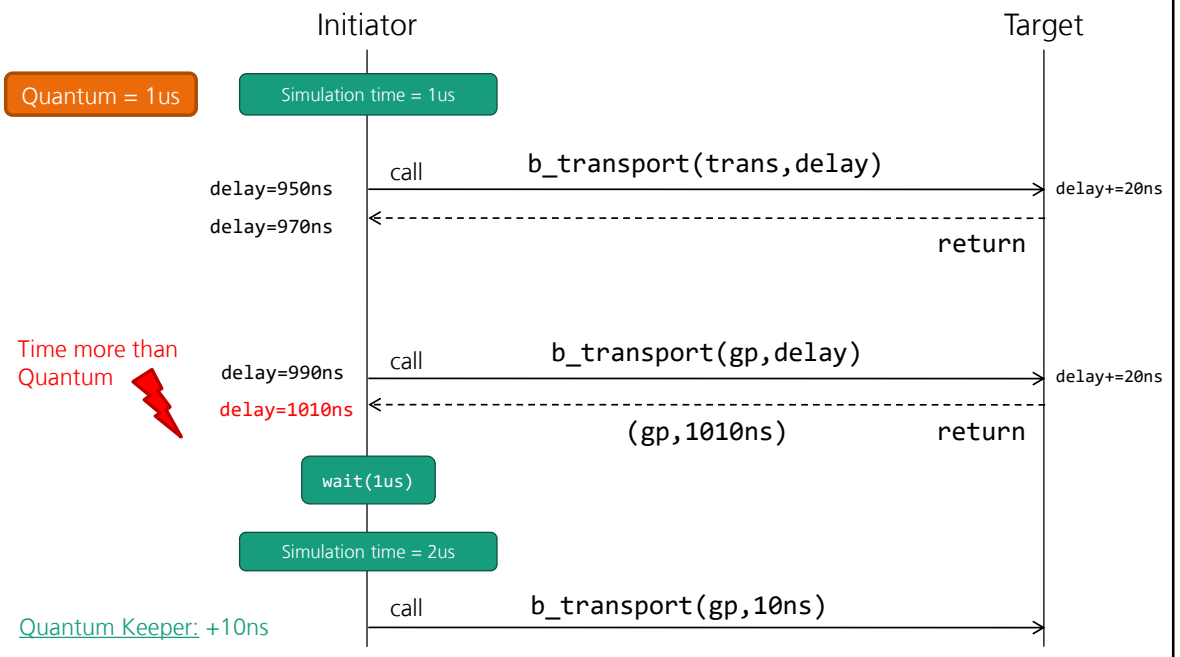


- Frequent context switches between processes has bad influence on sim.-speed
- In temporal decoupled simulation mode a process keeps control until a quantum is reached, then its switched to another process
- Processes can run ahead of time! Out-of-order execution! Synchronization!

Fraunhofer IESE

Your notes:

Blocking Interface (LT) with Quantum



Fraunhofer IESE

Your notes:

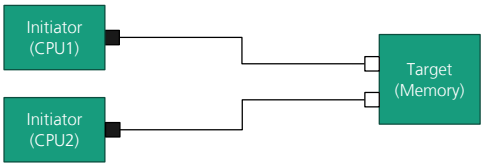
Quantum Keeper

SW

```
class Initiator: sc_module,
                 tlm::tlm_bw_transport_if<>
{
  private:
  tlm_utils::tlm_quantumkeeper quantumKeeper;

  public:
  tlm::tlm_initiator_socket<> iSocket;
  SC_CTOR(exampleInitiator) : iSocket("iSocket")
  {
    iSocket.bind(*this);
    SC_THREAD(process);
    quantumKeeper.set_global_quantum(
      sc_time(1,SC_US)
    );
    quantumKeeper.reset();
  }

  void process()
  {
    // Write to memory:
    for (int i = 0; i < 1024; i++) {
      tlm::tlm_generic_payload trans;
      unsigned char data = rand();
      trans.set_address(i);
      trans.set_data_length(1);
      trans.set_command(tlm::TLM_WRITE_COMMAND);
      trans.set_data_ptr(&data);

      sc_time delay = quantumKeeper.get_local_time();

      iSocket->b_transport(trans, delay);
      // Anotate the time of the target
      quantumKeeper.set(delay);

      // Consume internal computation time
      quantumKeeper.inc(sc_time(10,SC_NS));

      if(quantumKeeper.need_sync())
      {
        quantumKeeper.sync();
      }
    }
  }
  …
  // Dummy methods …
};
```

Static Method

Calls `wait()` internally

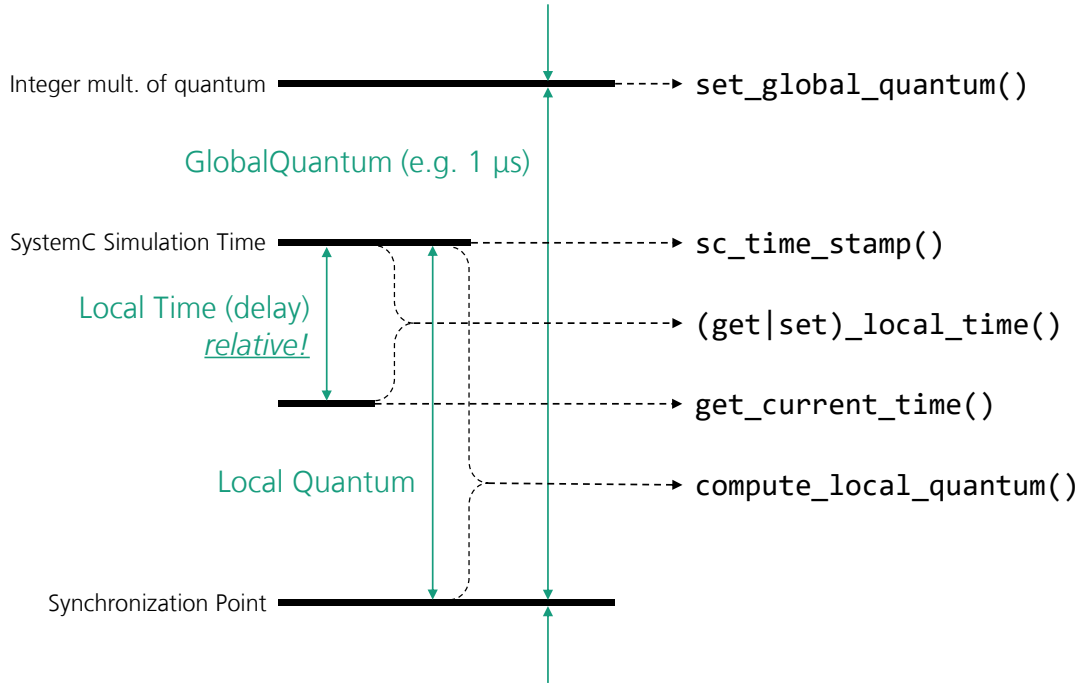


Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_quantum_keeper

Fraunhofer IESE

Your notes:

Quantum Keeper Variables and Methods

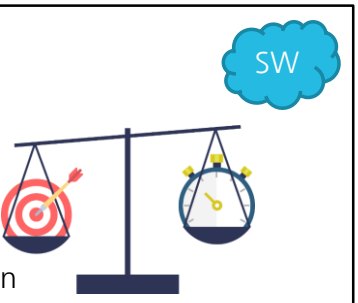


Fraunhofer IESE

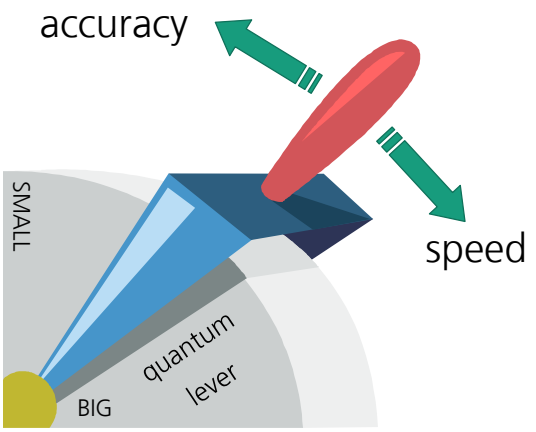
- There are three absolute time variables, namely the current simulation time as returned by sc_time_stamp(), the current time decoupled from simulation time, and the time of the next synchronization point.
- The global quantum is the time between two sync points.
- The local time is the time offset of the current time (as used by a temporally decoupled initiator) from the SystemC simulation time.
- The current time is the absolute time value as used by a temporally decoupled initiator.

Your notes:

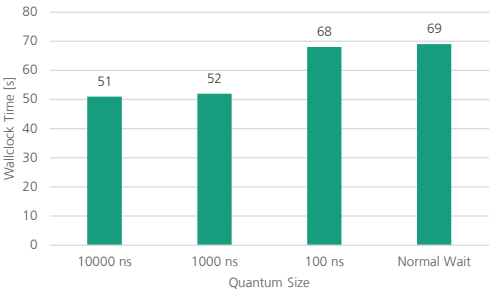
Quantum vs. Accuracy



- Quantum is user configurable
- Trade-off between simulation speed and accuracy
- The smaller the quantum, the more accurate the simulation
- If target uses `wait()` internally, it should set the `delay = SC_ZERO_TIME`



Our Artifacts Example (i.7, MacOS):

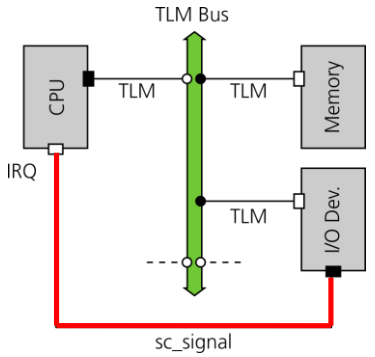


46

Fraunhofer IESE

Your notes:

A Closer Look on Functional Simulation Errors

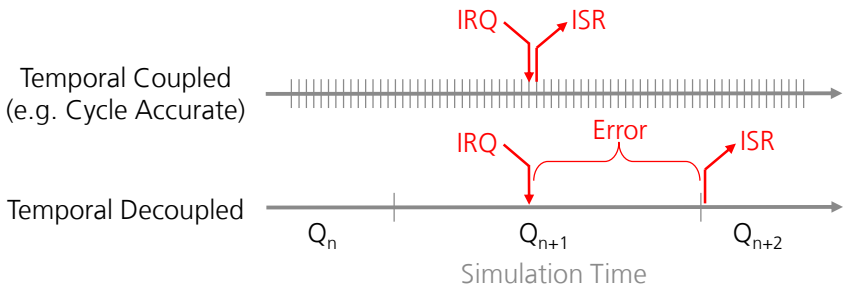


Temporal Coupled (e.g. Cycle Accurate)

- I/O Device makes an *Interrupt Request* (IRQ)
- The *Interrupt Service Routine* (ISR) will be called a few cycles later

Temporal Decoupled Simulation

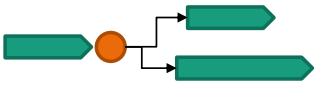
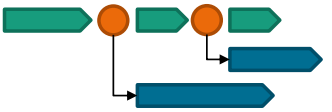
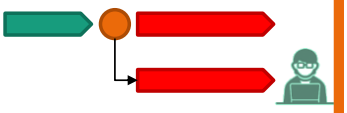
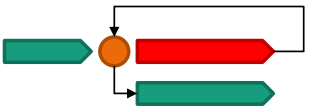
- I/O Device makes an IRQ
- The ISR will be called in the next Quantum



Fraunhofer IESE

Checkpointing

- The ability to save the state of a simulation and later pick up at the exact same point in time.

Save the boot	Parallelize Test Execution	Save for the Day
		
Avoid redoing work, save booted system for reuse and distribution	*Spin up additional parallel simulations during testing*	*Save work, shut down simulation, continue the next day*
Gear Shifting	**Debug SW / Model**	**Undo Target Actions**
		
Save the state from a fast model and continue in a detailed model	*Save state when bug happens, transfer to developer for analysis*	*Get back to a previous good state after sth. went wrong*

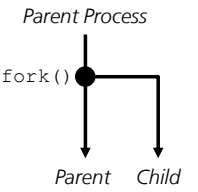
- Gläser et al. [4] presented in 2015 the idea of using checkpointing in order to rollback in simulation time and force an earlier synchronization to correct the occurred errors.

Source: Jacob Engblom, Intel, SystemC Evolution Day, 2017

[4] Temporal decoupling with error-bounded predictive quantum control. Georg Glaeser, Gregor Nitsche, Eckhard Hennig. Published in Forum on Specification and Design Languages (FDL) 2015

Fraunhofer IESE

The Good Old `fork()`



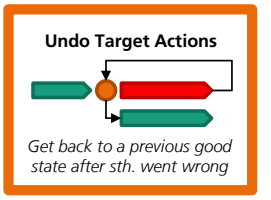
- `fork()` is a system call that allows a process in the OS to create a one-to-one copy of itself, called *child*.
- Supported by all OS: Linux, FreeBSD, macOS, …
- Modern OS do not duplicate the complete memory space of a process
- Instead they use the Copy-on-Write semantic:
  - The copy operation is deferred to the first write to a memory page
  - In other words: a memory page is only copied in the moment of the change

Can fork() be used as an efficient way for checkpointing in order to get an error free temporal decoupled simulation?

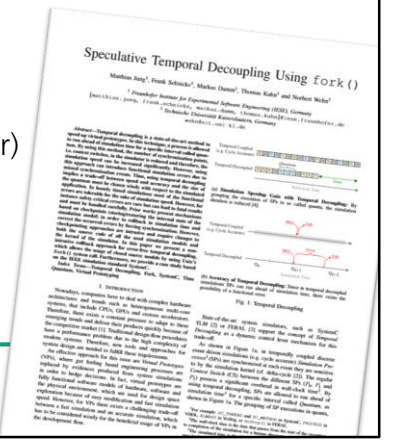
Fraunhofer IESE

Speculative Temporal Decoupling using `fork()`

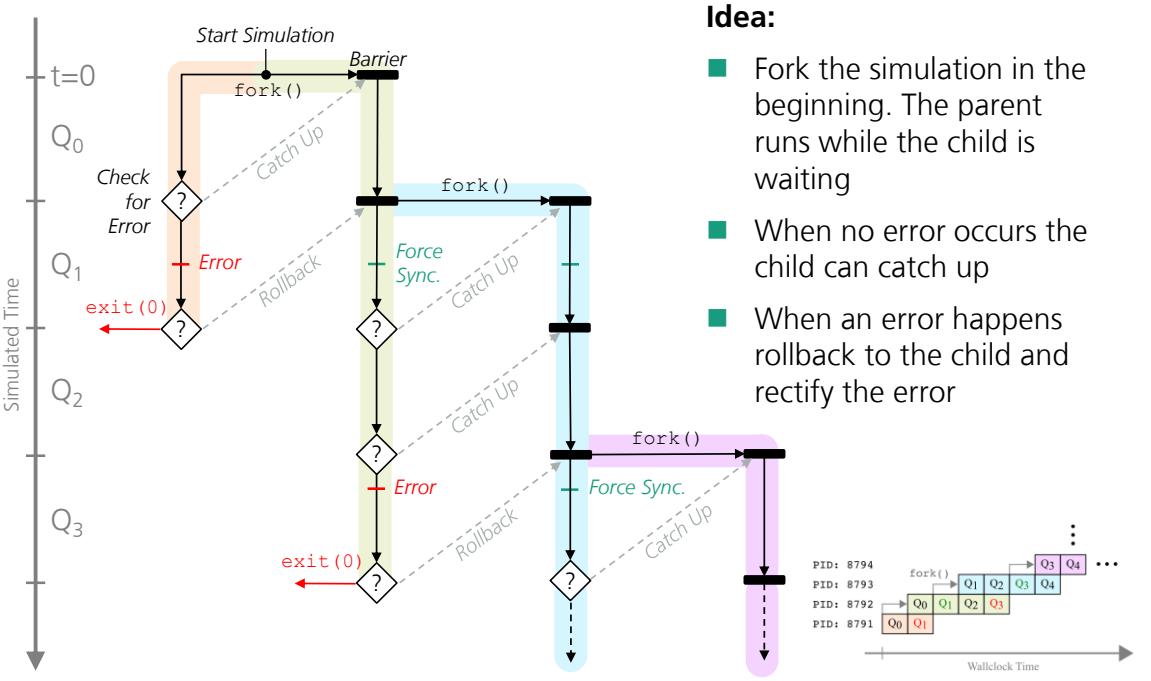
- Idea:
  - Use `fork()` to backup the simulation state
  - Execute the next quantum **speculatively**
  - In case of an error rollback in simulation time
  - Correct the timing error e.g. by temporary decreasing the quantum size



- Two approaches investigated:
  1. **Naïve Approach** (Forking at each quantum)
  2. **Lockstep Approach** (Forking only in case of an error)
- Synchronization of *parent* and *child* is done with `pipe` (acts like a barrier)
- Details of the implementation are in the paper


Speculative Temporal Decoupling Using fork()

Approach

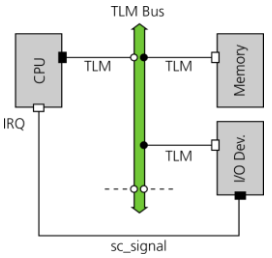


Idea:

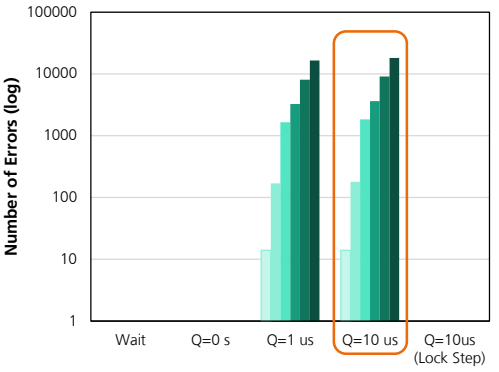
- Fork the simulation in the beginning. The parent runs while the child is waiting
- When no error occurs the child can catch up
- When an error happens rollback to the child and rectify the error

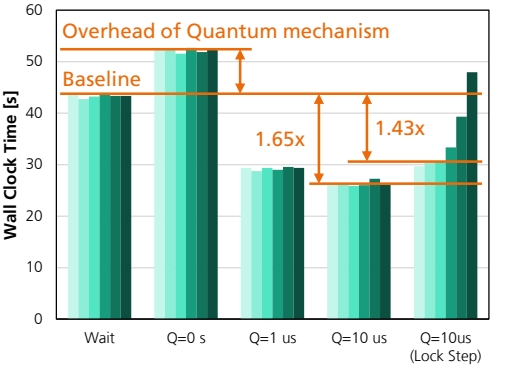
Fraunhofer IESE

Results for the Approach

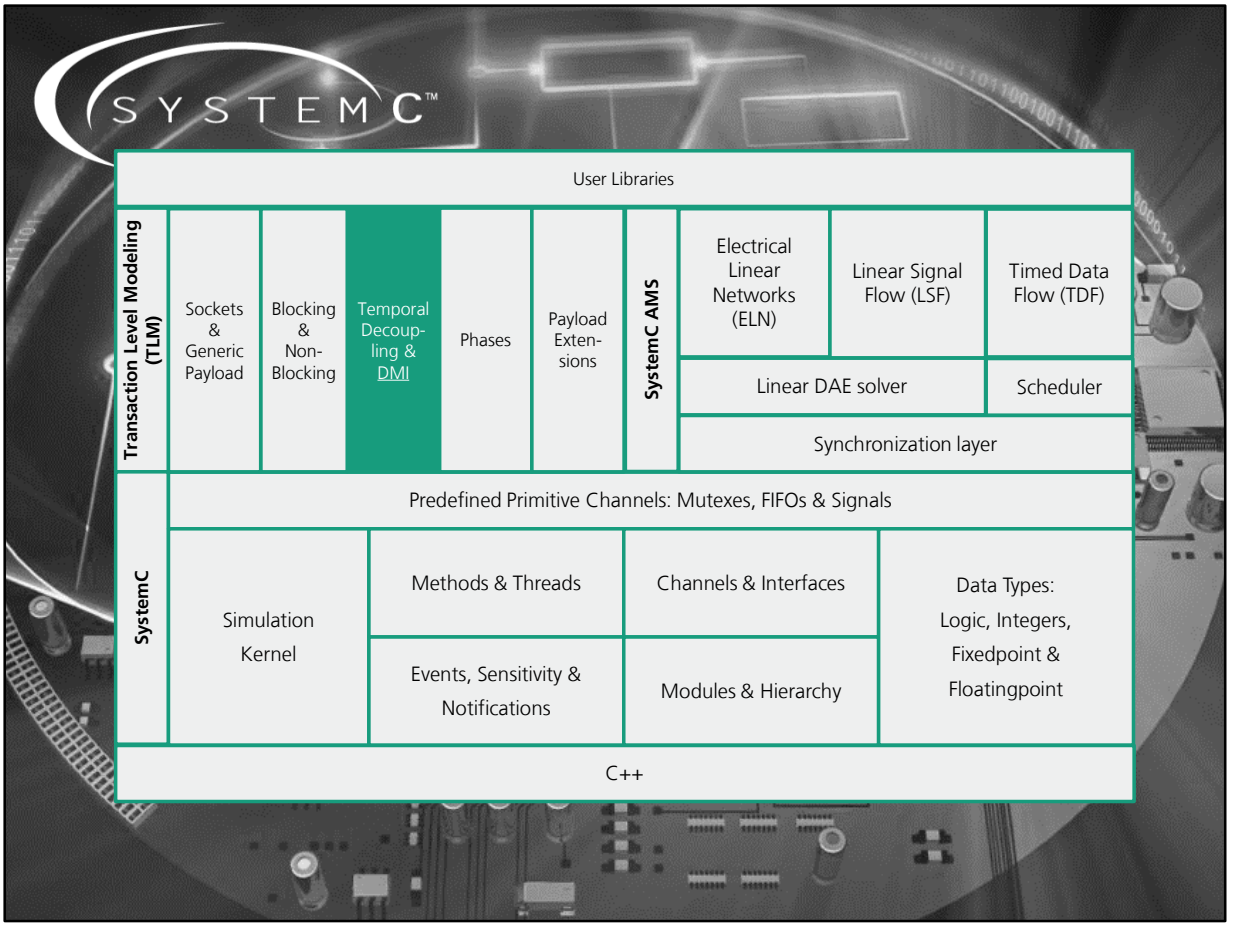


- Example System with one CPU and I/O Device
- Interrupt rates between 1s - 1ms
- Synchronization with `wait()`
- Synchronization with different quanta (0s, 1us, 10us)
- Errors = Number of missed IRQ events



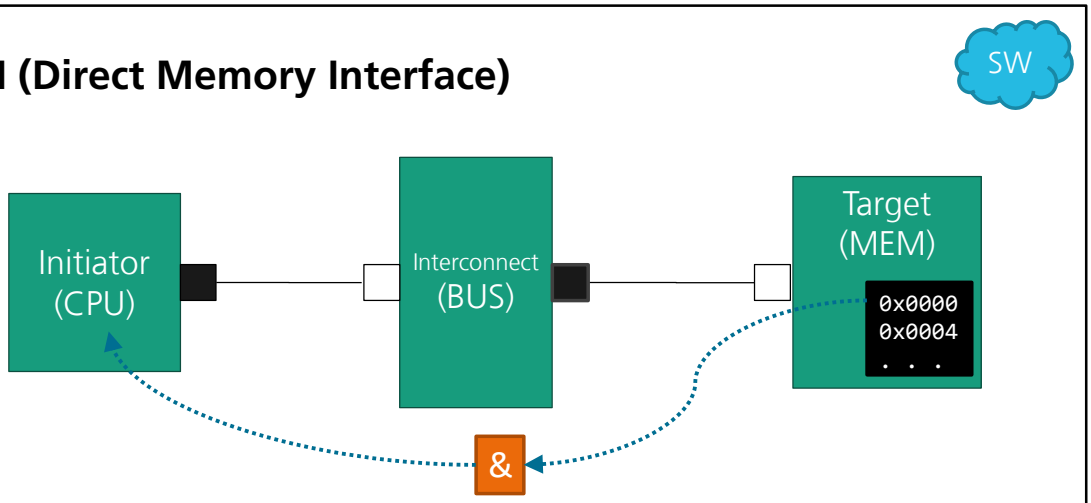


Fraunhofer IESE


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++

Your notes:

DMI (Direct Memory Interface)



- Bypasses the Interconnects (e.g. Bus or Cache) and all socket & transport calls!
- Gives an initiator a pointer to memory region in the target
- Target can give a hint to initiator that DMI is available
- Uses also generic payload, can use also extensions
- Target can invalidate DMI regions
- Gives higher simulation speed, e.g. for booting an OS …

Fraunhofer IESE

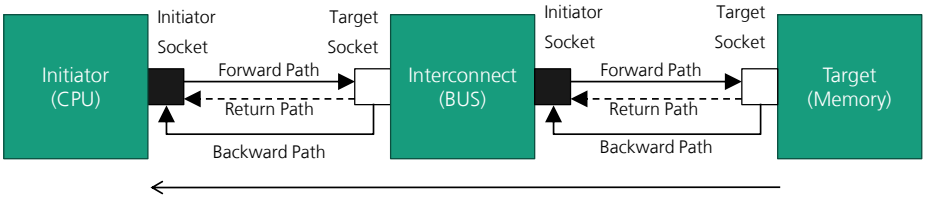
The main reason for DMI is simulation speed. DMI allows the transport interface to be bypassed by giving an initiator a direct pointer to an area of memory in a target. Instead of calling transport for each read and write operation, the initiator can simply access the memory directly. For operations other than read or write, it is possible to add extensions to the DMI transaction using the standard generic payload extension mechanism. It is the responsibility of the initiator to honor any invalidation calls received from the target

Your notes:

DMI (Direct Memory Interface)

SW

```
bool get_direct_mem_ptr(tlm_generic_payload trans, tlm_dmi dmiData);
```




```
void invalidate_direct_mem_ptr(uint64 start, uint64 end);
```

- Same routing as e.g. `b_transport`
- Class `tlm_dmi`:
  - `unsigned char* dmi_ptr`
  - `uint64 start_address`
  - `uint64 end_address`
  - `dmi_access_e granted_access`
  - `sc_time read_latency`
  - `sc_time write_latency`

Fraunhofer IESE

Your notes:

DMI Initiator

SW

```
class Initiator: sc_module, tlm::tlm_bw_transport_if<> {
  bool dmi // set to false in the constructor;
  tlm::tlm_dmi dmiData;
  ...
  void process() {
    for (int i = 0; i < 16; i++)
    {
      tlm::tlm_generic_payload trans;
      unsigned char data = rand();
      trans.set_address(i);
      trans.set_data_length(1);
      trans.set_command(tlm::TLM_WRITE_COMMAND);
      trans.set_data_ptr(&data);
      sc_time delay = sc_time(0, SC_NS);

      if ( dmi == true
           && i >= dmiData.get_start_address()
           && i <= dmiData.get_end_address())
      {
        if( trans.get_command() == tlm::TLM_READ_COMMAND
            && dmiData.is_read_allowed())
        {
          memcpy(&data,
          dmiData.get_dmi_ptr() + i
              - dmiData.get_start_address(),
          trans.get_data_length());
          delay += dmiData.get_read_latency();
        }
        else if( trans.get_command()==tlm::TLM_WRITE_COMMAND
          && dmiData.is_write_allowed())
        {
          memcpy(dmiData.get_dmi_ptr() + i
              - dmiData.get_start_address(),
          &data,
          trans.get_data_length());
          delay += dmiData.get_write_latency();
        }
```

DMI start here

Normal `b_transport`

```
      } else {
        iSocket->b_transport(trans, delay);

        if(trans.is_dmi_allowed() == true) {
          dmiData.init(); // Reset DMI descriptor
          dmi = iSocket->get_direct_mem_ptr(trans, dmiData);
        }
      }
      wait(delay);
    } // end for
    sc_stop();
  } // end process

  void invalidate_direct_mem_ptr(sc_dt::uint64 start_range,
                                 sc_dt::uint64 end_range)
  {
    dmi = false;
  }

  // Dummy methods
  ...
};
```

Get DMI Hint!

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

Fraunhofer IESE

Your notes:

DMI Interconnect


```
class exampleInterconnect : sc_module, tlm::tlm_bw_transport_if<>, tlm::tlm_fw_transport_if<>
{
    public:
    tlm::tlm_initiator_socket<> iSocket;
    tlm::tlm_target_socket<> tSocket;

    SC_CTOR(exampleInterconnect) {
        tSocket.bind(*this);
        iSocket.bind(*this);
    }

    void b_transport(tlm::tlm_generic_payload &trans, sc_time &delay) {
        delay = delay + sc_time(40, SC_NS);
        iSocket->b_transport(trans, delay);
    }

    bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmi_data) {
        bool dmi = iSocket->get_direct_mem_ptr(trans, dmi_data);

        dmi_data.set_read_latency( dmi_data.get_read_latency() + sc_time(40, SC_NS));
        dmi_data.set_write_latency( dmi_data.get_write_latency() + sc_time(40, SC_NS));

        return dmi;
    }

    void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
    {
        tSocket->invalidate_direct_mem_ptr(start_range, end_range);
    }

    // Dummy methods …
};
```

Forwarding DMI request on forward and backward path

Fraunhofer IESE

Your notes:

DMI Target


```
class exampleTarget : sc_module, tlm::tlm_fw_transport_if<> {
    unsigned char mem[512];
    public:
    tlm::tlm_target_socket<> tSocket;

    SC_CTOR(exampleTarget) : tSocket("tSocket") {
        tSocket.bind(*this);
        SC_THREAD(invalidateProcess);
    }
    void invalidateProcess() {
        while(true) {
            wait(500, SC_NS);
            tSocket->invalidate_direct_mem_ptr(0,511);
        }
    }
    void b_transport(tlm::tlm_generic_payload &trans, sc_time &delay) {
        ...
        trans.set_dmi_allowed( true );
    }
    bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmi_data) {
        std::cout << "get_direct_mem_ptr called" << std::endl;
        dmi_data.set_dmi_ptr(mem);
        dmi_data.set_start_address(0);
        dmi_data.set_end_address(511);
        dmi_data.set_read_latency(sc_time(40, SC_NS));
        dmi_data.set_write_latency(sc_time(40, SC_NS));
        dmi_data.allow_read_write();
        return true;
    }
    // Dummy methods ...
};
```

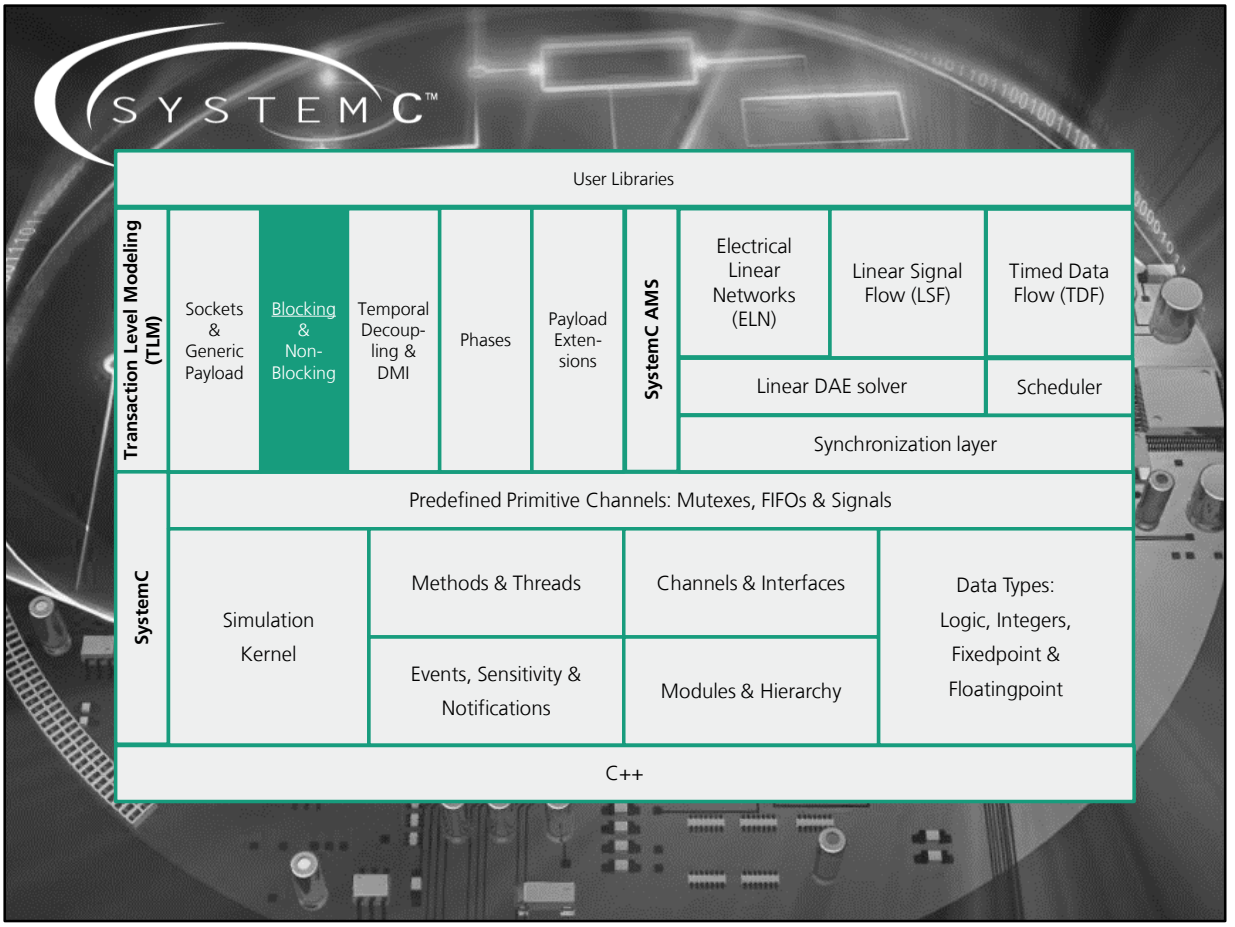
In this example the DMI access is invalidated every 500 ns, which is just an artificial example

Give Initiator a hint that DMI is possible

Configure DMI object with all relevant information

Fraunhofer IESE

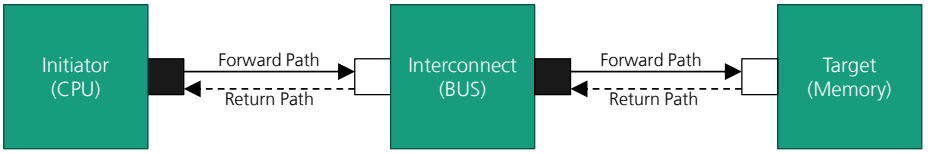
Your notes:


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++

Your notes:

Debug Transport `transport_dbg`

```
unsigned int transport_dbg(tlm_generic_payload trans);
```



- Gives an initiator debug access to memory in a target
- Similar to b_transport
  - Different: delay free, no waits, no event notifications
  - Uses generic payload
  - Same routing as b_transport
- Used for initialization, e.g. for bootloading

Fraunhofer IESE

Your notes:

Debug Transport `transport_dbg`

```
class Initiator : sc_module, tlm::tlm_bw_transport_if<> {
    ...
    void process() {
        ... // End of simulation:
        dumpMemory();
    }

    void dumpMemory()
    {
        unsigned char buffer[64];

        tlm::tlm_generic_payload trans;
        trans.set_address(0);
        trans.set_read();
        trans.set_data_length(64);
        trans.set_data_ptr(buffer);

        unsigned int n = iSocket->transport_dbg(trans);

        for(unsigned int i = 0; i < n; i++) {
            std::cout << std::hex
                      << std::setfill('0')
                      << std::setw(2)
                      << (unsigned int)buffer[i];

            if((i+1)%8 == 0) {
                std::cout << std::endl;
            }
        }
    }
};
```

```
class Target : sc_module, tlm::tlm_fw_transport_if<>
{
  ...
  void b_transport(... &trans, ... &delay)
  {
    ...
  }

  unsigned int transport_dbg(&trans)
  {
    if (trans.get_address() >= 1024) {
      return 0;
    }

    if(trans.get_command() == tlm::TLM_WRITE_COMMAND)
    {
       memcpy(&mem[trans.get_address()],
              trans.get_data_ptr(),
              trans.get_data_length());
    } else /* tlm::TLM_READ_COMMAND */ {
      memcpy(trans.get_data_ptr(),
             &mem[trans.get_address(),
             trans.get_data_length());
    }
    return trans.get_data_length();
  }
};
```

Try code on github:

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

63

Fraunhofer IESE

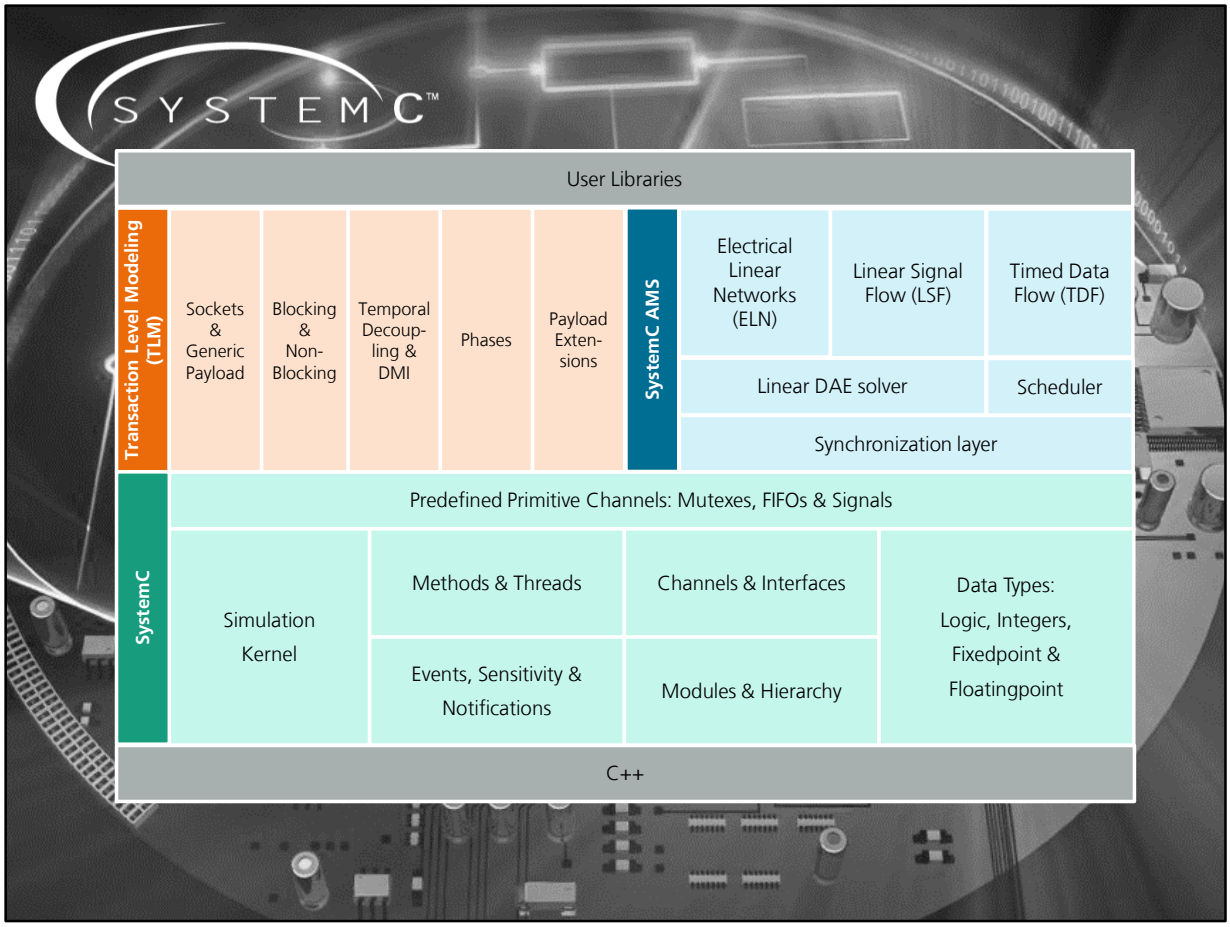
Your notes:

SystemC and Virtual Prototyping

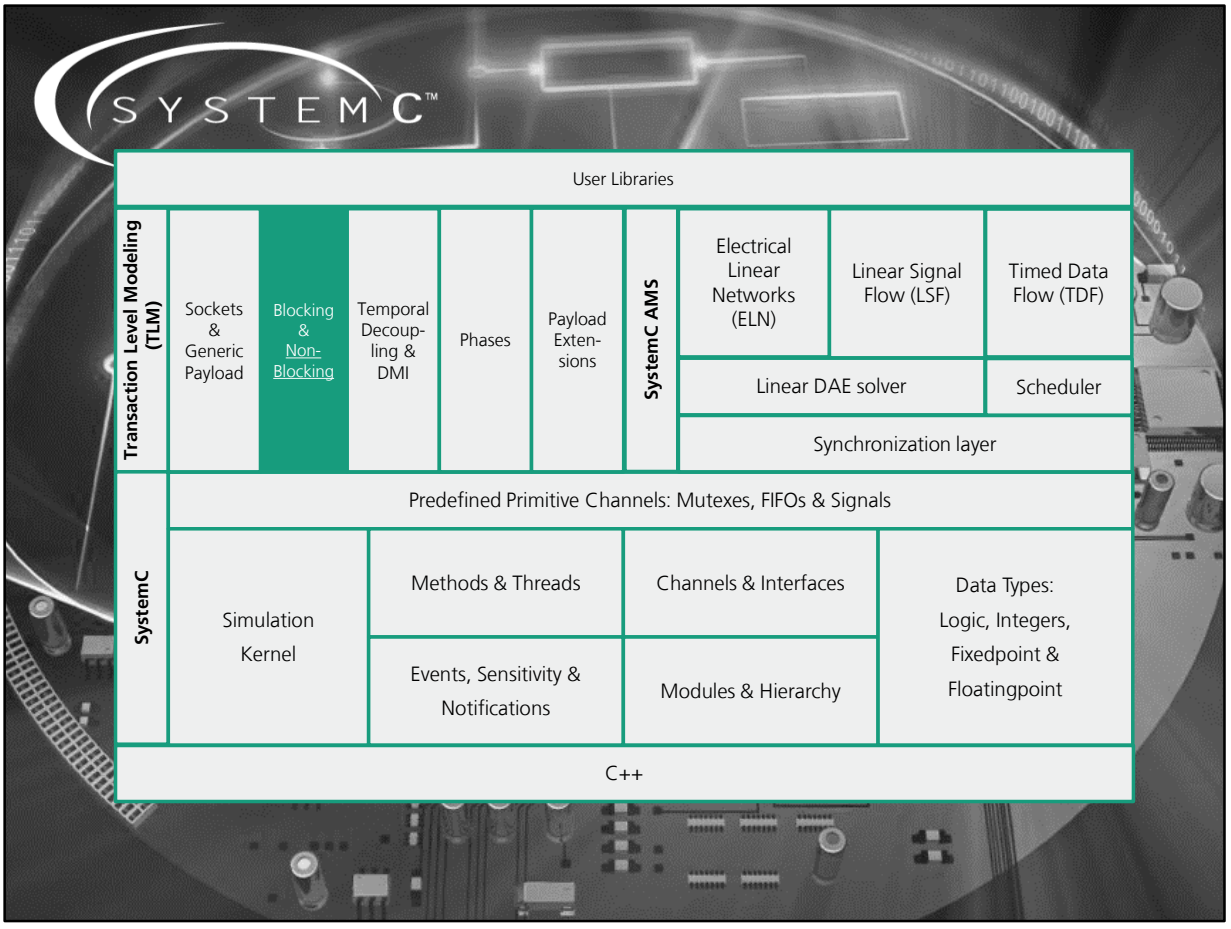
Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

Fraunhofer IESE

Your notes:

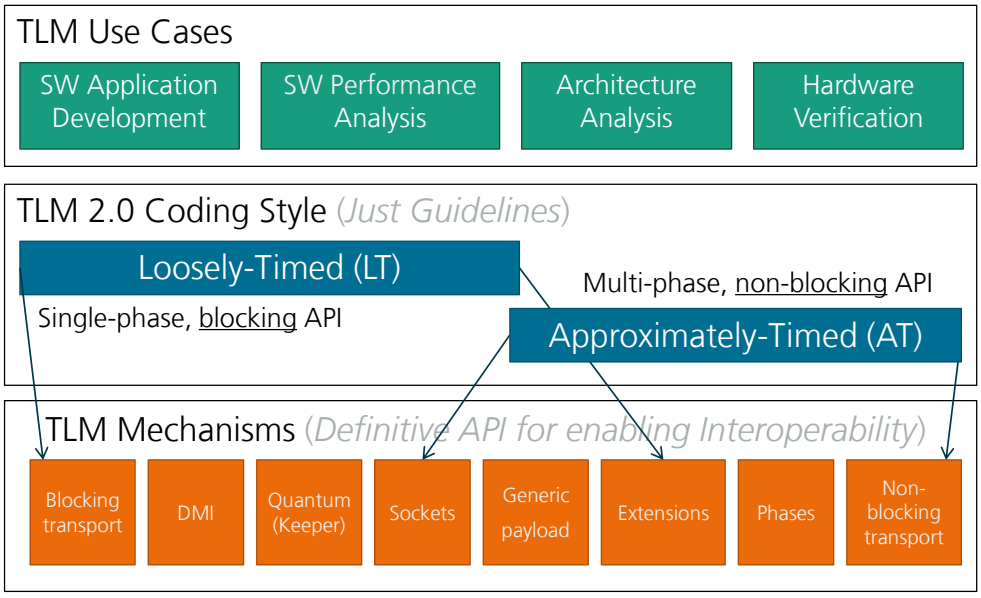

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:


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++

Your notes:

Recap: TLM Coding Styles and Mechanisms



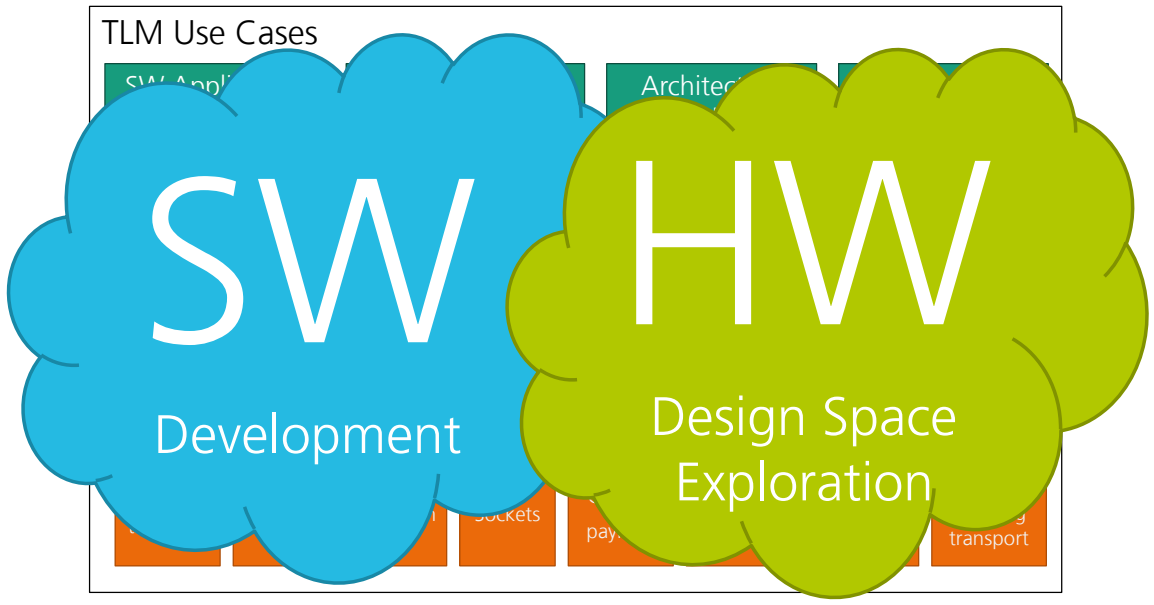
4

Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Your notes:

Recap: TLM Coding Styles and Mechanisms



5

Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

Your notes:

Recap: Coding Styles in TLM

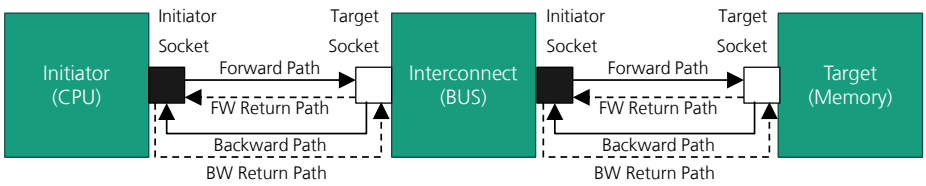
- **Loosely-Timed (LT):**
  - As fast as possible
  - Sufficient timing detail to boot OS and run multicore systems and to develop SW or drivers
  - Processes can run ahead of simulation time (temporal decupling)
  - Each transaction completes in one blocking function call
  - Usage of Direct Memory Interface (DMI) e.g. for boot process

- **Approximately-Timed (AT):**
  - Accurate enough for performance modelling
  - Sufficient for architectural HW design space exploration
  - Processes run in lockstep with simulation time
  - Each transaction has usually 4 timing points i.e. 4 function calls (extensible if required, also less possible); non-blocking behavior
  - More detailed than LT and therefore also slower than LT

Fraunhofer IESE

Your notes:

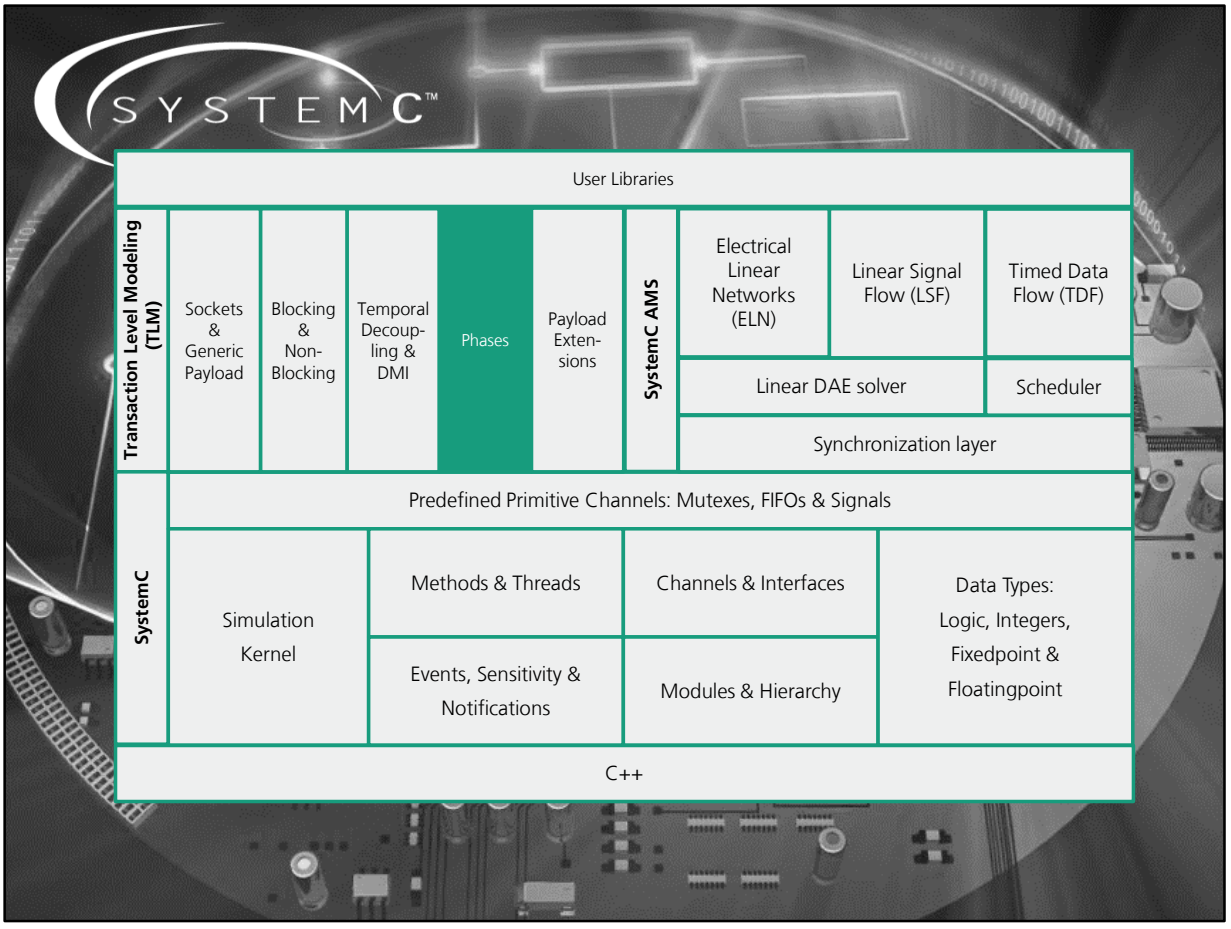
Initiators, Targets and Interconnect



- References to the object are passed along the forward and backward paths:
  - LT uses Forward and Return Path
  - AT uses Forward, Backward, FW Return Path, BW Return Path
- AT uses non-blocking transport
- Time is handeled with Payload Event Queues (PEQs)
- Allows modelling of O-O-O Cores and Backpressure
- Base Protocol

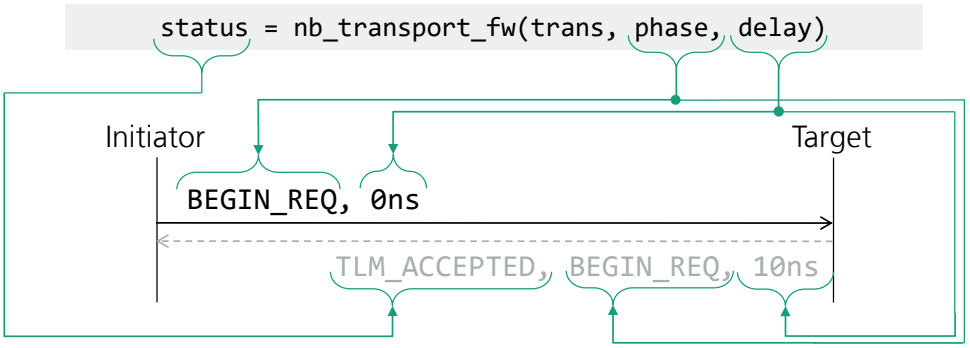
Fraunhofer IESE

Your notes:


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++

Your notes:

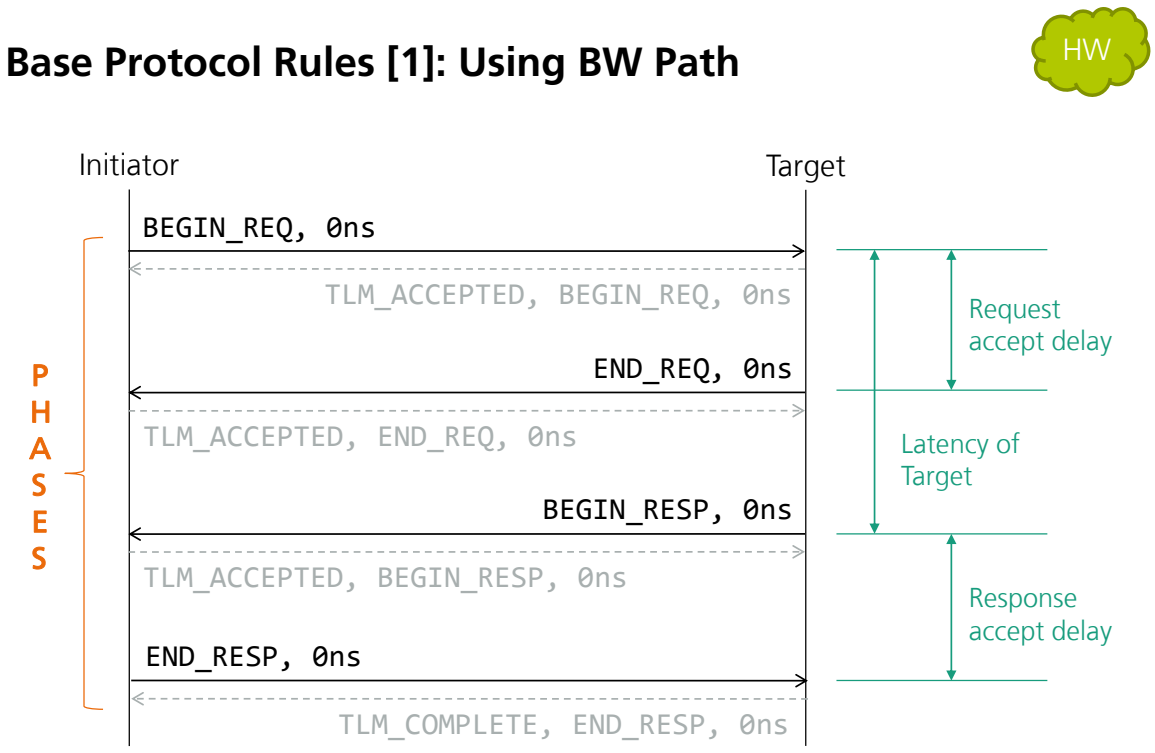
Non-Blocking Transport (AT) Base Protocol



Fraunhofer IESE

Your notes:

Base Protocol Rules [1]: Using BW Path



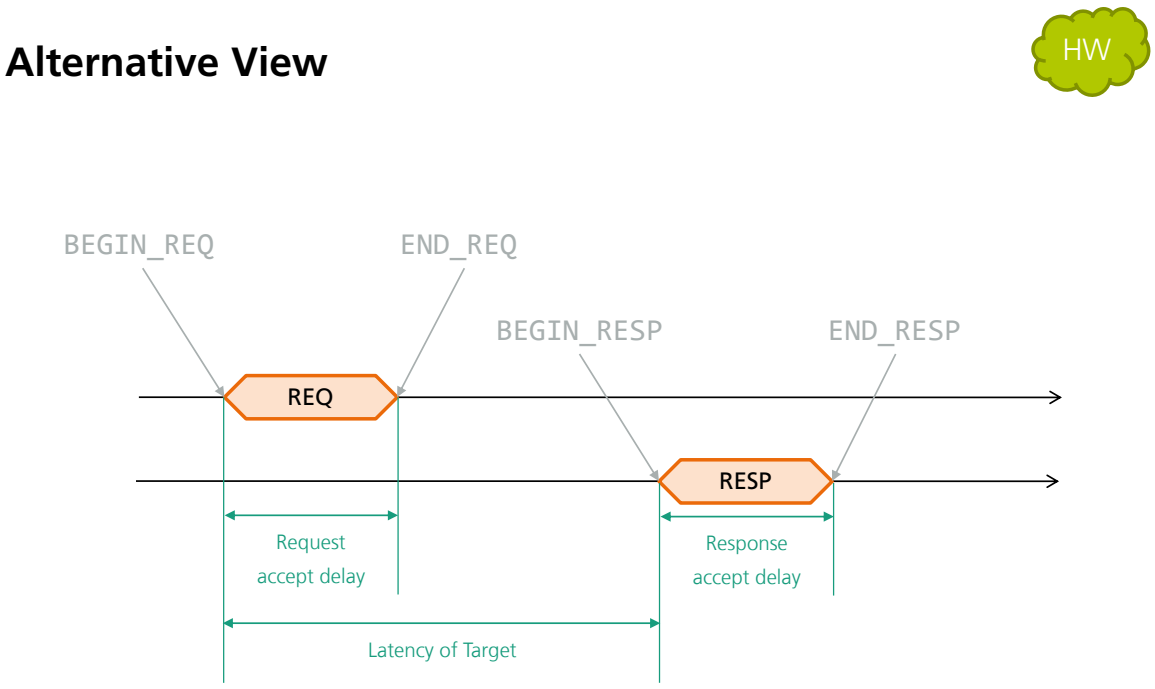
Fraunhofer IESE

An initiator, and only an initiator, may call `b_transport`. The target can return immediately, or can suspend (by calling `wait`) and return later. However, calling `wait` statements in targets should be avoided because a wait statement will result in a context switch of the simulator, which decreases the simulation speed.

The same transaction object could be reused from one call to the next, but the two calls would count as separate transactions.

Your notes:

Alternative View



11

Fraunhofer IESE

Your notes:

Base Protocol Rules: Using BW Path

- Base Protocol phases
  - `BEGIN_REQ` → `END_REQ` → `BEGIN_RESP` → `END_RESP`
  - Must occur in increasing simulation time order
  - Phases must change with each call
- nb_transport_fw must not call nb_transport_bw directly and vice versa (PEQ!)
- Generic Payload memory management rules (See TLM Advanced)
- Extensions must be ignorable (See TLM Advanced)
- Target should handle mixed b_transport / nb_transport

Fraunhofer IESE

Your notes:

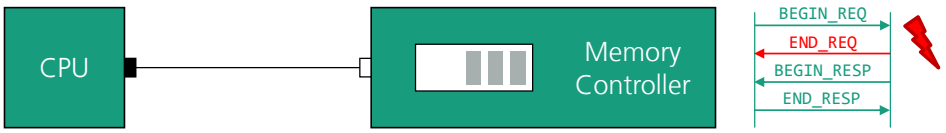
Base Protocol Rules: The Exclusion Rule

- Initiators and targets must honor the *Exclusion Rule*:
  - An Initiator must not send a new request (`BEGIN_REQ`) until it has received the `END_REQ` from the previous transaction
  - An target must not send the next `BEGIN_RESP` until it has received the `END_RESP` from the previous transactions
- The exclusion rules enable flow control like back pressure:
  - E.g. if an input buffer of a target is full the target can defer the sending of `END_REQ` for the transaction that filled the buffer, until the buffer has available space again
  - An RTL ready signal can be modeled by deferring `END_REQ`
  - However, since it is non-blocking, the target can do something else, e.g. sending

Fraunhofer IESE

Your notes:

Modelling of Backpressure

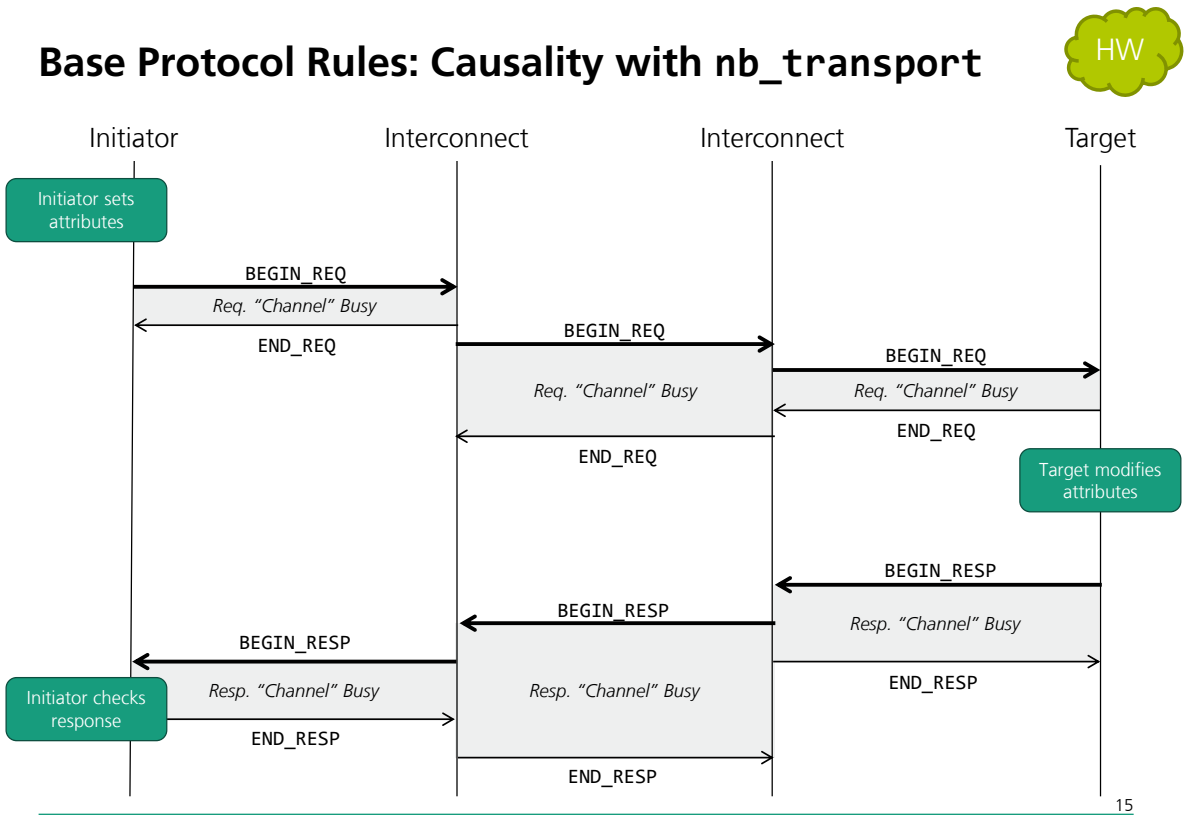


- The exclusion rules enable flow control like back pressure:
  - E.g. if an input buffer of a target is full the target can defer the sending of `END_REQ` for the transaction that filled the buffer, until the buffer has available space again
  - An RTL ready signal can be modeled by deferring `END_REQ`
  - However, since it is non-blocking, the target can do something else, e.g. sending
- Also Response exclusion rule must be honored!

Fraunhofer IESE

Your notes:

Base Protocol Rules: Causality with `nb_transport`



15

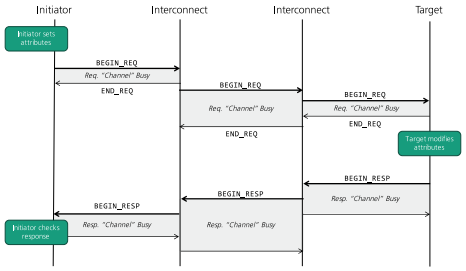
Fraunhofer IESE

Your notes:

Base Protocol Rules: Causality with `nb_transport`

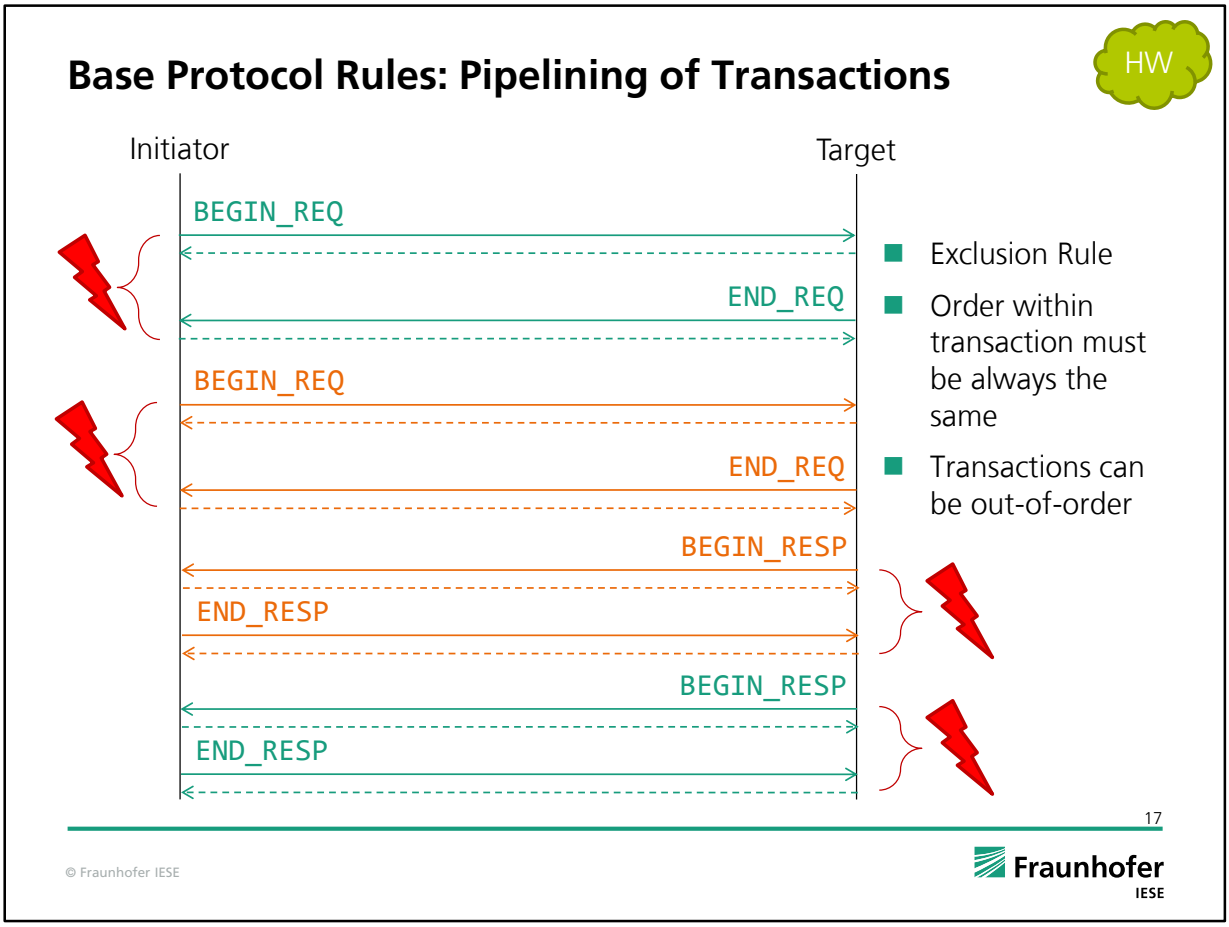
HW

- `BEGIN_REQ` and `BEGIN_RESP` are propagated from end-to-end
- `END_REQ` and `END_RESP` are not propagated → they are local to each hop
- The `END_REQ` and `END_RESP` phases indicate that the requests and response "channels" are no longer busy and may be used for the next request or response respectively
- Initiator should not check the transaction payload until it has received the response



Fraunhofer IESE

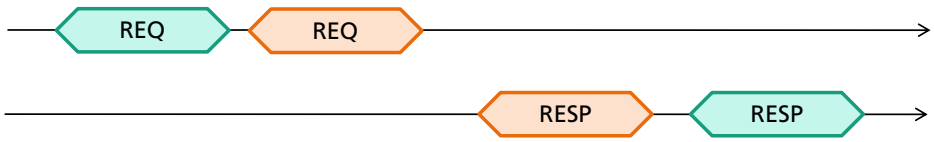
Your notes:



For **nb_transport_(fw|bw),** the timing annotation on successive calls for a given transaction must be strictly non-decreasing. For separate transactions, the mutual order is unconstrained, i.e. out-of-order transaction ordering is allowed.

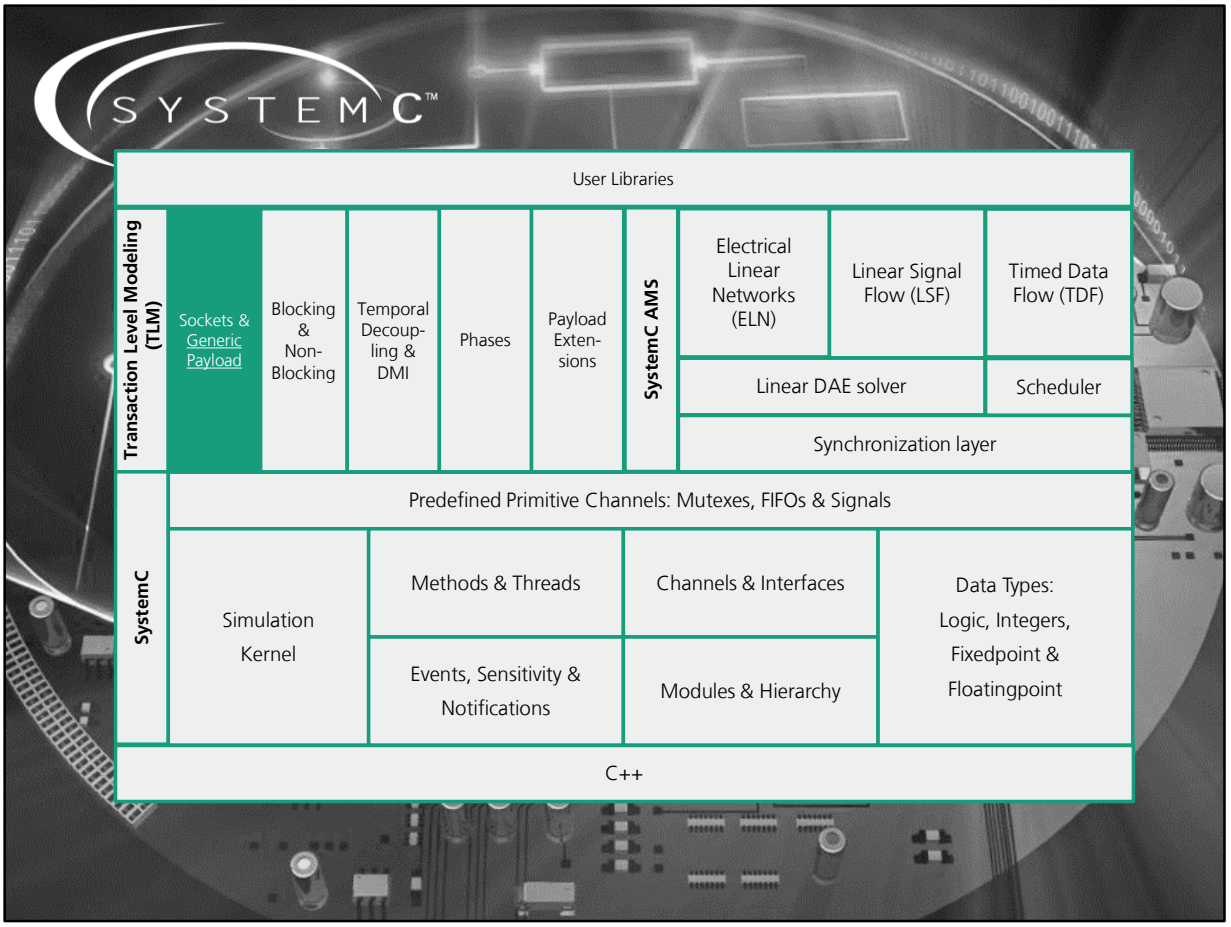
Your notes:

Alternative View



Fraunhofer IESE

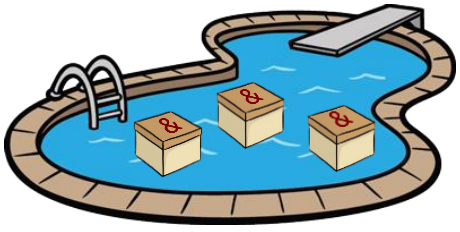
Your notes:


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++

Your notes:

Generic Payload Pools (a.k.a Memory Management)

- Allocating of generic payload objects consumes wall-clock time
- Idea: do not allocate for each transaction a new generic payload → **Reuse**
  - A *Memory Manager* handles the generic payload objects in a pool
  - Before sending a transaction a payload object is `allocate`d from the pool
  - Modules that send (or receive) this payload object must increase a reference count (`acquire`), which signalizes the memory manager that this object is still in use.
  - If a module is finished with the payload object, the reference count is decreased (`release`)
  - If the reference count is 0 the payload is freed and goes back into the pool.
  - If all transactions in the pool are in use a new one is generated



Fraunhofer IESE

Your notes:

Memory Manager in TLM

- The memory manager is not part of the SystemC TLM Standard
- The TLM Standard provides only an interface class:

```
class tlm_mm_interface
{
    public:
    virtual void free(tlm::tlm_generic_payload*) = 0;
    virtual ~tlm_mm_interface(){}
}
```

- The implementation of the memory manager is up to the end user
- Virtual Platform tools like Synopsys Platform Architect have clever implementations

Fraunhofer IESE

Your notes:

Example Memory Manager

Use code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_memory_manager

```
class MemoryManager : public tlm::tlm_mm_interface {
    private:
    unsigned int numberOfAllocations;
    unsigned int numberOfFrees;
    std::vector<tlm::tlm_generic_payload*> freePayloads;

    public:
    MemoryManager(): numberOfAllocations(0), numberOfFrees(0) {}

    ~MemoryManager(){
        for(tlm::tlm_generic_payload* payload: freePayloads) {
            delete payload;
            numberOfFrees++;
        }
    }

    tlm::tlm_generic_payload* MemoryManager::allocate(){
        if(freePayloads.empty()) {
            numberOfAllocations++;
            return new tlm::tlm_generic_payload(this);
        } else {
            tlm::tlm_generic_payload* result = freePayloads.back();
            freePayloads.pop_back();
            return result;
        }
    }

    void free(tlm::tlm_generic_payload* payload) {
        payload->reset(); //clears all fields and extensions
        freePayloads.push_back(payload);
    }
};
```

Transaction Pool implemented as `std::vector`

Cleanup transaction pool

Allocate new payload

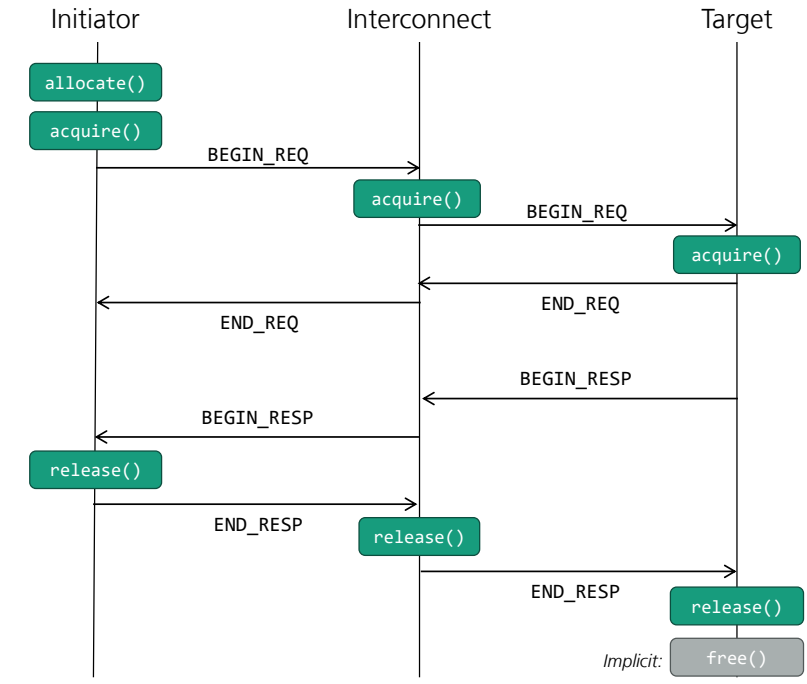
Reuse Payload and remove from pool

Add transaction back to the pool

Fraunhofer IESE

Your notes:

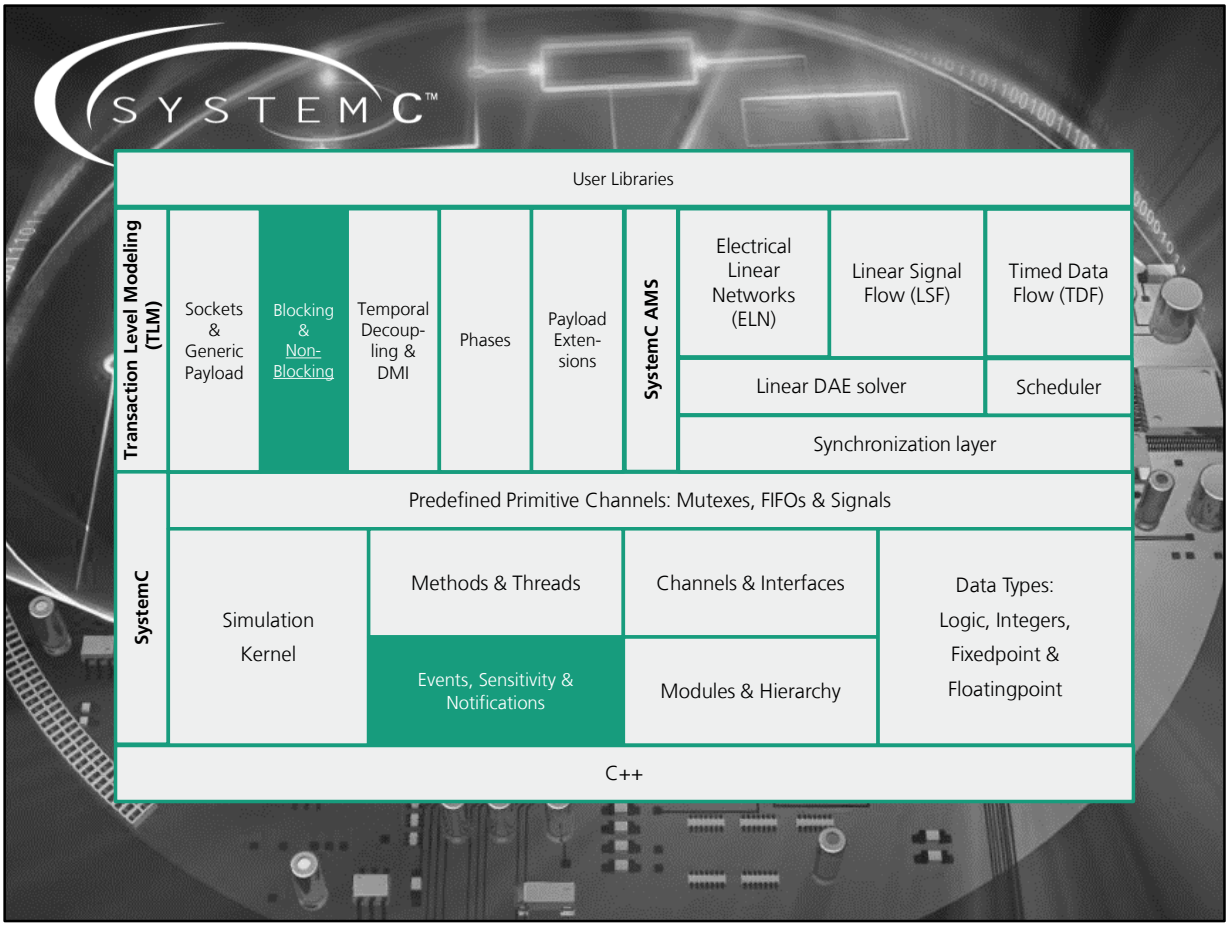
Usage of Memory Manager



- `b_transport` a.k.a. LT may use a memory manager
- `nb_transport` a.k.a. AT must use a memory manager

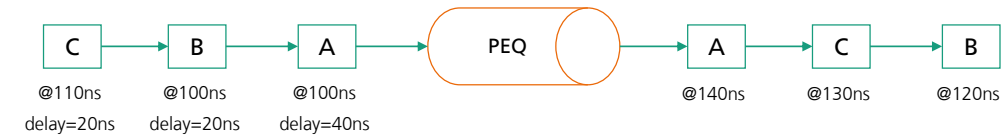
Fraunhofer IESE

Your notes:


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++

Your notes:

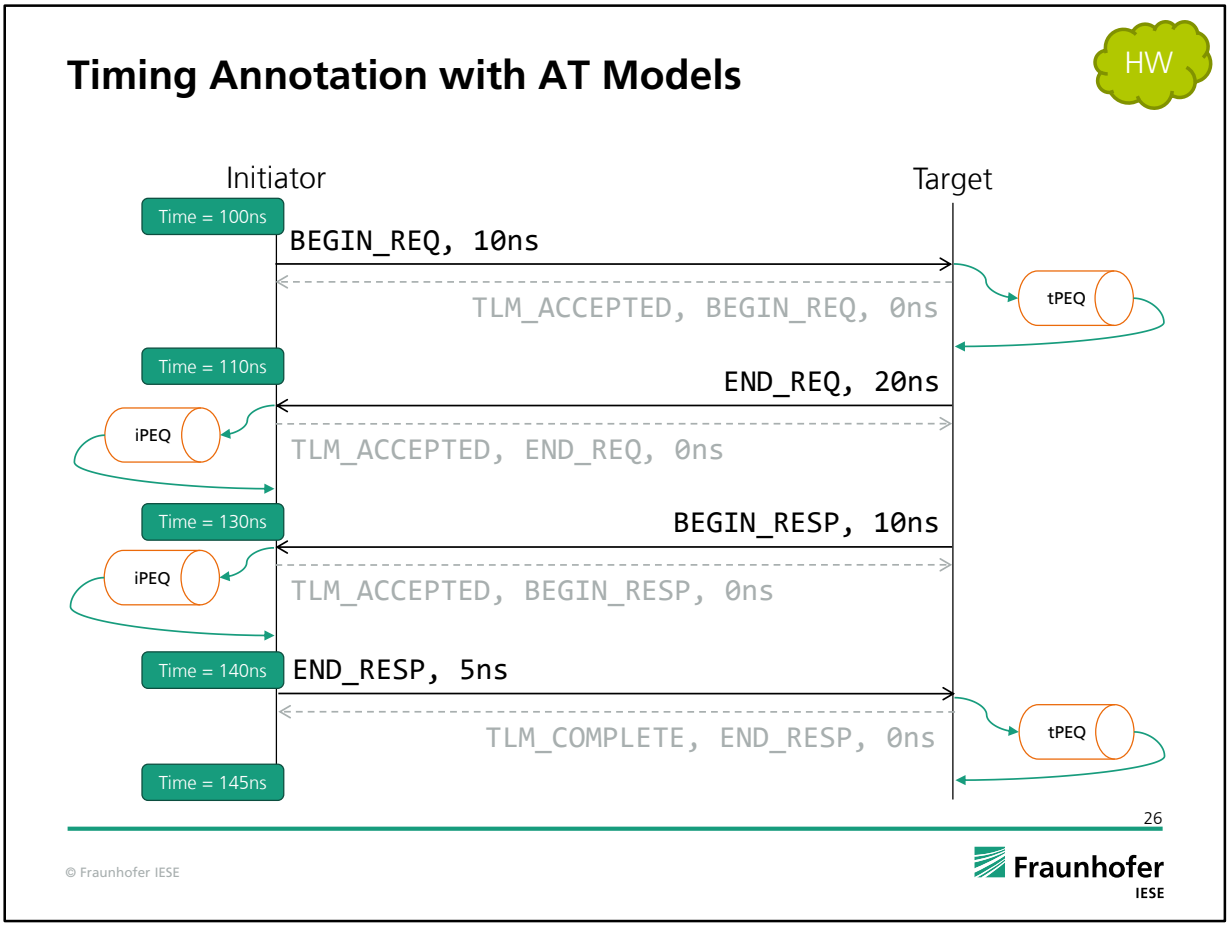
The Payload Event Queue



- AT models usually synchronize incoming calls with the simulation time
- Timing annotation is similar to b_transport:
  - *Hey target, please pretend that you received this message 10ns in the future*
- An AT component is usually not calling `wait()` statements for synchronizing with time, it rather posts the incoming transaction into a *Payload Event Queue* (PEQ). The PEQ internally synchronizes with the simulation time and calls a callback function in order to process the transaction when the time is reached.
- A component receives a transaction that should be processed in the future, so it puts the transaction into a PEQ in order to process it when time is reached

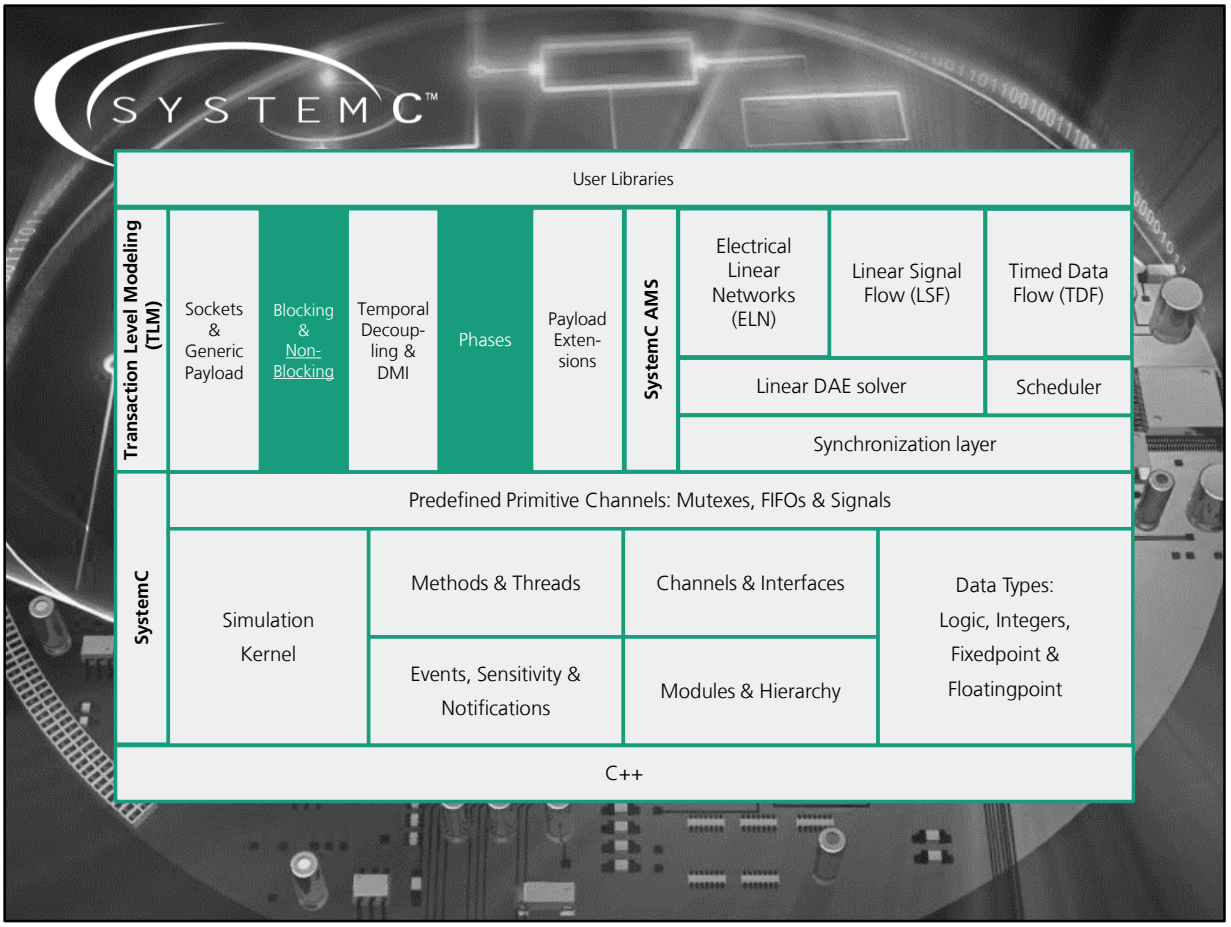
Fraunhofer IESE

Your notes:



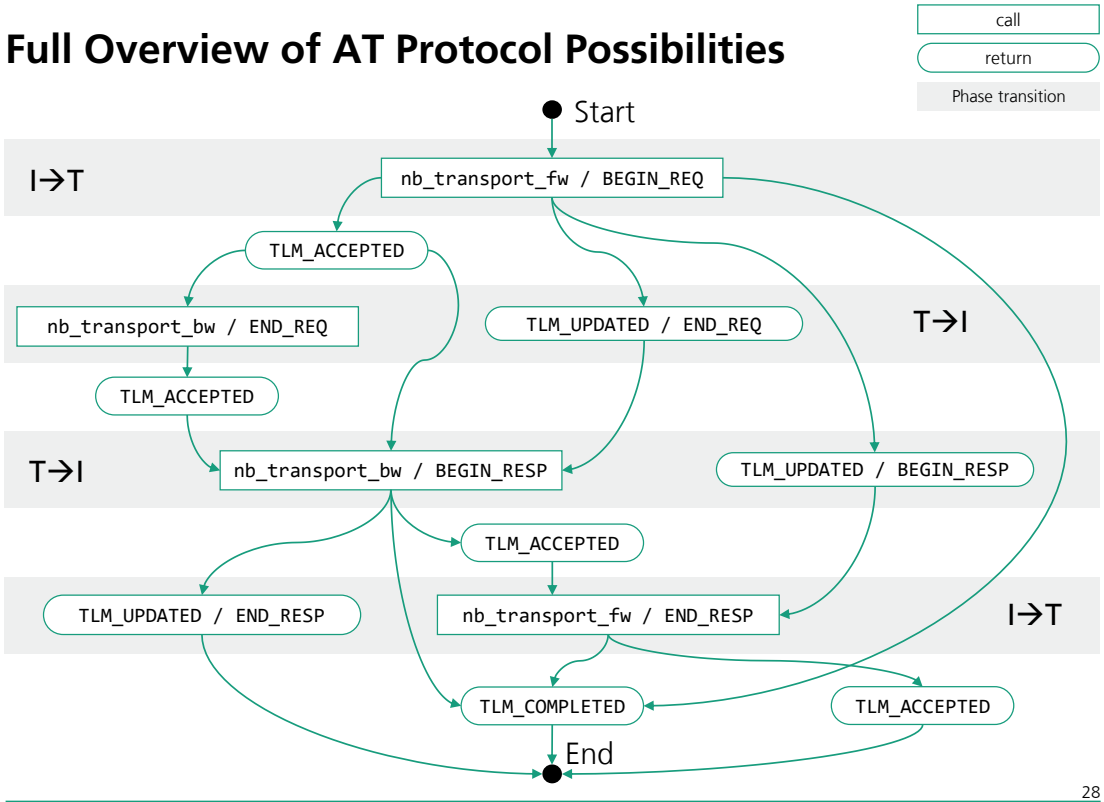
In AT models it is usual to synchronize incoming `nb_transport` calls with the simulation time. In the message sequence chart above, an initiator sends `BEGIN_REQ` with a timing annotation of 10 ns. The target returns `TLM_ACCEPTED` and is obliged to behave as if it had received the transaction at `sc_time_stamp() + 10ns`. An AT component is usually not calling wait statements for synchronizing with time, it rather posts the incoming transaction into a *Payload Event Queue* (PEQ). The payload event queue internally synchronizes with the simulation time (`wait`) and calls a callback function in order to process the transaction when the time is reached. In other words, a component receives a transaction that should be processed in the future, so it puts the transaction into a queue such that it is actually processed in the future.

Your notes:


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++

Your notes:

Full Overview of AT Protocol Possibilities

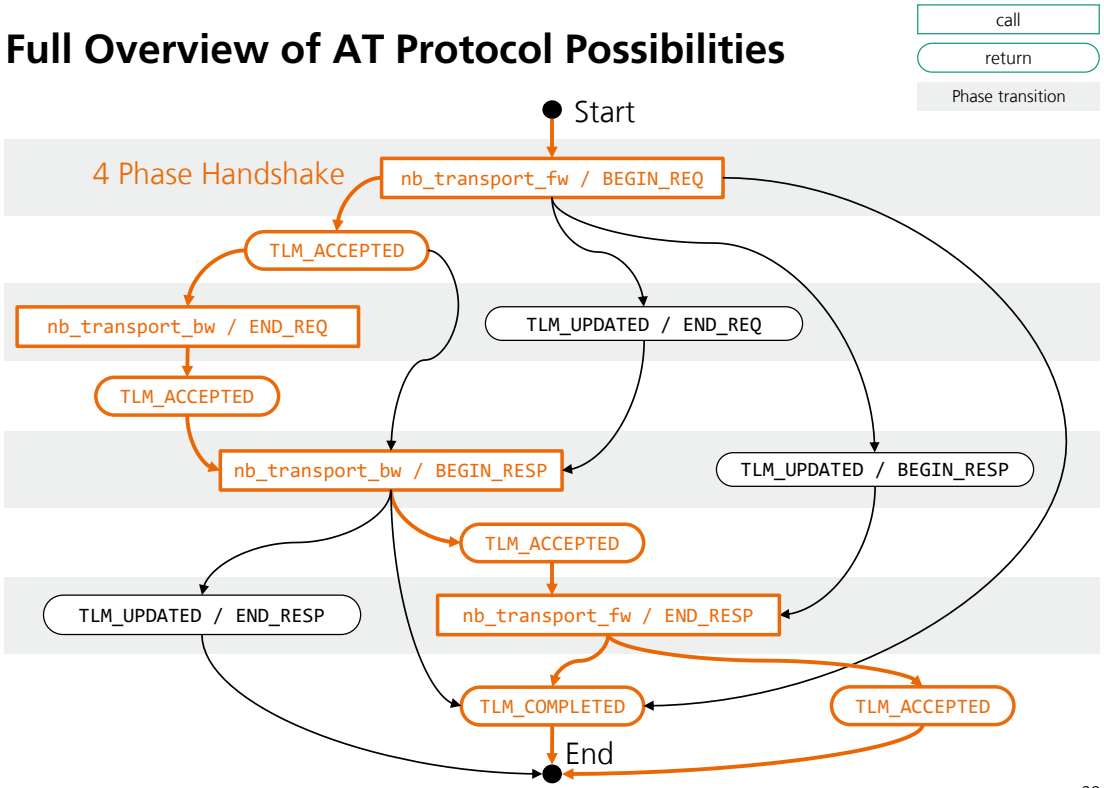


28

Fraunhofer IESE

Your notes:

Full Overview of AT Protocol Possibilities

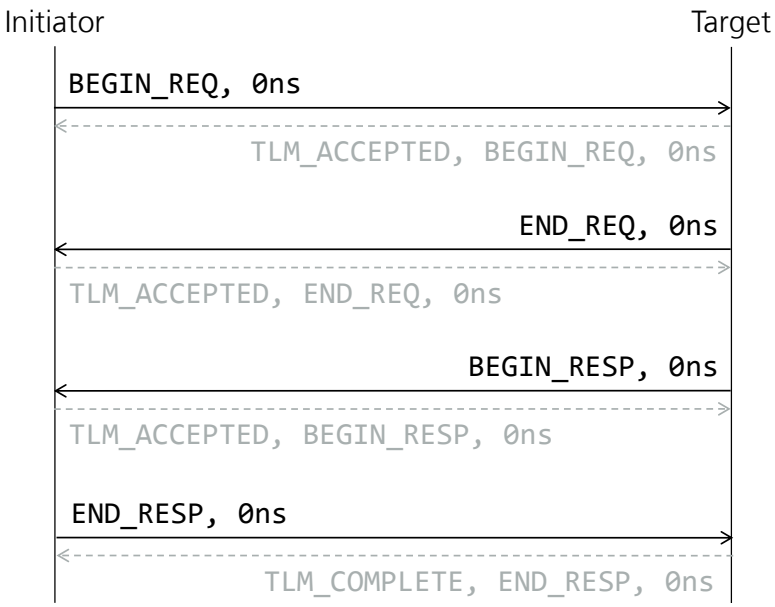


29

Fraunhofer IESE

Your notes:

Base Protocol Rules [1]: 4 Phase Handshake



Fraunhofer IESE

Your notes:

4 Phase Handshake: Initiator

HW

```
class Initiator: public sc_module, public tlm::tlm_bw_transport_if<> {
    public:
    tlm::tlm_initiator_socket<> socket;
    MemoryManager mm;
    int data[16];
    tlm::tlm_generic_payload* requestInProgress;
    sc_event endRequest;
    tlm_utils::peq_with_cb_and_phase<Initiator> peq;

    SC_CTOR(Initiator): socket("socket"),
                        requestInProgress(0),
                        peq(this, &Initiator::peqCallback)
    {
        socket.bind(*this);
        SC_THREAD(process);
    }

    protected:
    void process() {
        tlm::tlm_generic_payload* trans;
        tlm::tlm_phase phase;
        sc_time delay;

        for (int i = 0; i < 100; i++) {
            trans = mm.allocate();
            trans->acquire();

            trans->set_command(...);
            ...
            trans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

            if (requestInProgress) {
                wait(endRequest);
            }
            requestInProgress = trans;
            phase = tlm::BEGIN_REQ;
            delay = ...;

            tlm::tlm_sync_enum status;
            status = socket->nb_transport_fw( *trans, phase, delay );
            ...
            wait(randomDelay());
        }
    }
```

Generate a sequence of random transactions

Grab a new transaction from the memory manager pool

BEGIN_REQ/END_REQ exclusion rule

```
    virtual tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload& trans,
                                               tlm::tlm_phase& phase,
                                               sc_time& delay)
    {
        peq.notify(trans, phase, delay);
        return tlm::TLM_ACCEPTED;
    }

void peqCallback(tlm::tlm_generic_payload& trans,
                 const tlm::tlm_phase& phase)
    {
        if (phase == tlm::END_REQ || ...)
        {
            requestInProgress = 0;
            endRequest.notify();
        }
        else  if (phase == tlm::BEGIN_RESP)
        {
            checkTransaction(trans);

            tlm::tlm_phase fw_phase = tlm::END_RESP;
            sc_time delay = sc_time(...);

            socket->nb_transport_fw( trans, fw_phase, delay );

            trans.release();
        }
        else if (phase == tlm::BEGIN_REQ || phase == tlm::END_RESP)
        {
            SC_REPORT_FATAL(name(), "Illegal transaction phase received");
        }

    }
};
```

Queue the transaction into the peq until the annotated time has elapsed

Payload event queue callback

Wake-up suspended main process

Do something with transaction

Allow MM to free the transaction object

Fraunhofer IESE

4 Phase Handshake: Target


```
class Target: public sc_module, public tlm::tlm_fw_transport_if<> {
    public:
    tlm::tlm_target_socket<> socket;
    tlm::tlm_generic_payload* transactionInProgress;
    sc_event targetDone;
    bool responseInProgress;
    tlm::tlm_generic_payload* nextResponsePending;
    tlm::tlm_generic_payload* endRequestPending;
    tlm_utils::peq_with_cb_and_phase<Target> peq;

    SC_CTOR(Target) : socket("socket"),
        transactionInProgress(0),
        responseInProgress(false),
        nextResponsePending(0),
        endRequestPending(0),
        peq(this, &Target::peqCallback)
    {
        socket.bind(*this);

        SC_METHOD(executeTransactionProcess);
        sensitive << targetDone; dont_initialize();
    }
    ...
    tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload& trans,
                                       tlm::tlm_phase& phase,
                                       sc_time& delay)
    {
        peq.notify( trans, phase, delay);
        return tlm::TLM_ACCEPTED;
    }

    void peqCallback(tlm::tlm_generic_payload& trans,
                     const tlm::tlm_phase& phase)
    {
        sc_time delay;

        if(phase == tlm::BEGIN_REQ) {
        trans.acquire();

            if (!transactionInProgress) {
                sendEndRequest(trans);
            }
            else {
                endRequestPending = &trans;
            }
        }
```

Increment the transaction reference count

Put back-pressure on initiator by deferring END_REQ

```
        else if (phase == tlm::END_RESP) {
            ...
            transactionInProgress = 0;

            responseInProgress = false;

            if (nextResponsePending) {
                sendResponse(*nextResponsePending);
                nextResponsePending = 0;
            }

            if (endRequestPending) {
                sendEndRequest(*endRequestPending);
                endRequestPending = 0;
            }

        }
        else // tlm::END_REQ or tlm::BEGIN_RESP
        {
            SC_REPORT_FATAL(name(), "Illegal transaction phase received");
        }
    }

    void sendEndRequest(tlm::tlm_generic_payload& trans)
    {
        tlm::tlm_phase bw_phase;
        sc_time delay;

        bw_phase = tlm::END_REQ;
        delay = ...; // Accept delay

        tlm::tlm_sync_enum status;
        status = socket->nb_transport_bw( trans, bw_phase, delay );

        delay = delay + randomDelay();
        targetDone.notify( delay );

        assert(transactionInProgress == 0);
        transactionInProgress = &trans;
    }
    ...
```

Flag must only be cleared when END_RESP is sent

Target itself is now clear to issue the next BEGIN_RESP

unblock the initiator by issuing END_REQ

Queue internal event to mark beginning of response

Fraunhofer IESE

4 Phase Handshake: Target


```
    void executeTransactionProcess()
    {

        executeTransaction(*transactionInProgress);

        if (responseInProgress)
        {
            ...
            nextResponsePending = transactionInProgress;
        }
        else
        {
            sendResponse(*transactionInProgress);
        }
    }


    void sendResponse(tlm::tlm_generic_payload& trans)
    {
        tlm::tlm_sync_enum status;
        tlm::tlm_phase bw_phase;
        sc_time delay;

        responseInProgress = true;
        bw_phase = tlm::BEGIN_RESP;
        delay = SC_ZERO_TIME;
        status = socket->nb_transport_bw( trans, bw_phase, delay );

        ...

        trans.release();
    }
...
};
```

Method process that runs on targetDone event

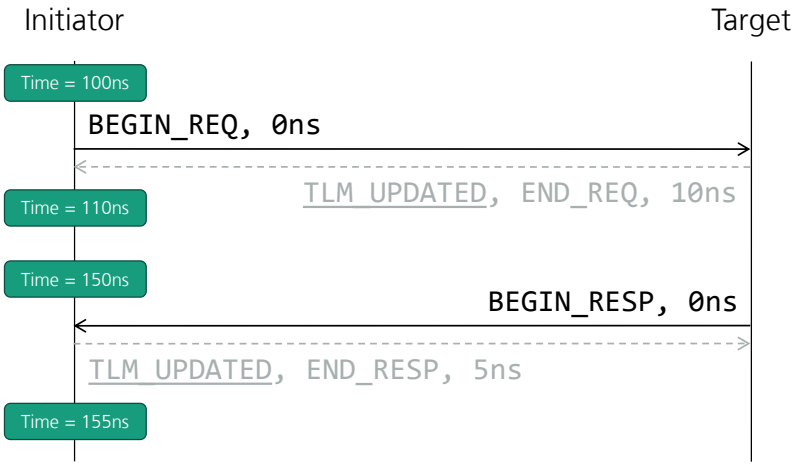
Target must honor BEGIN_RESP/END_RESP exclusion rule

Function for sending the response

Tell the memory manager to free transaction

Fraunhofer IESE

Base Protocol Rules [2]: Using the Return Paths

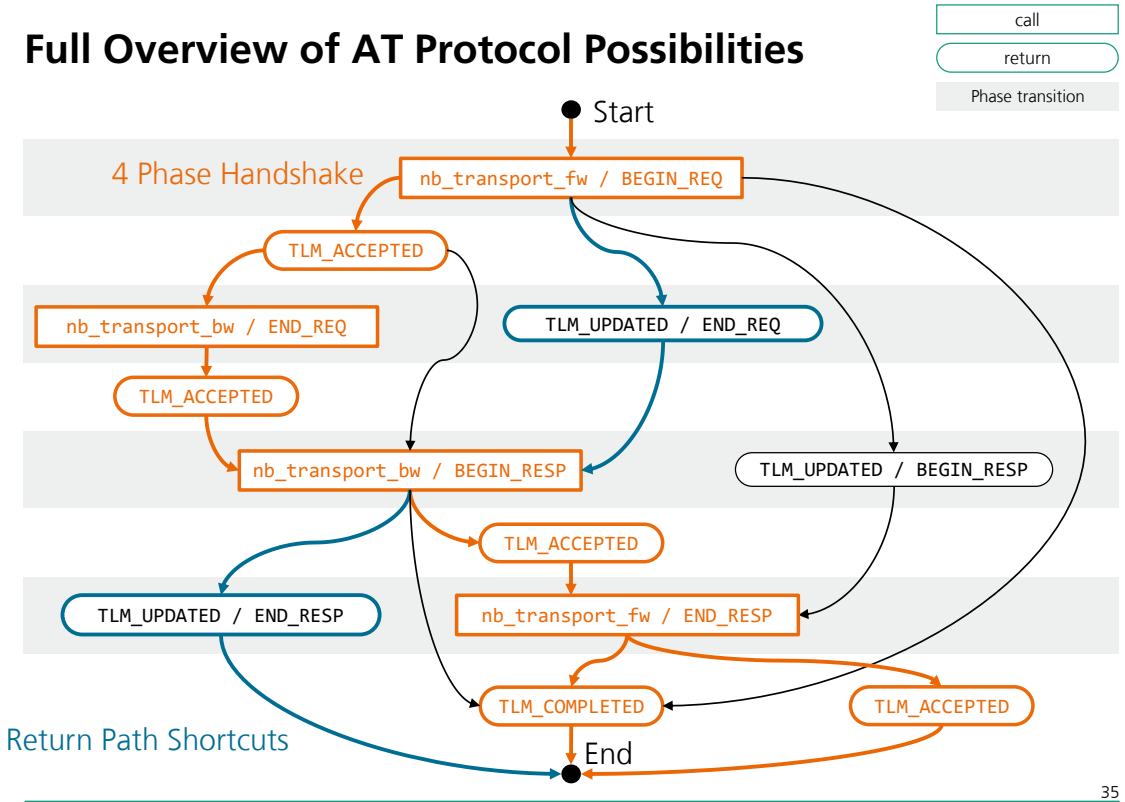


- The FW return path can be used as an alternative to the BW path
- The BW return path can be used as an alternative to the FW path
  - State must be set to `TLM_UPDATED`
- Timing must be increased

Fraunhofer IESE

Your notes:

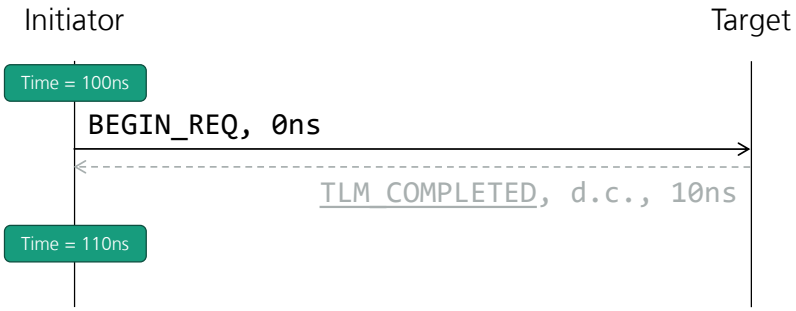
Full Overview of AT Protocol Possibilities



Fraunhofer IESE

Your notes:

Base Protocol Rules [4]: Early Completion

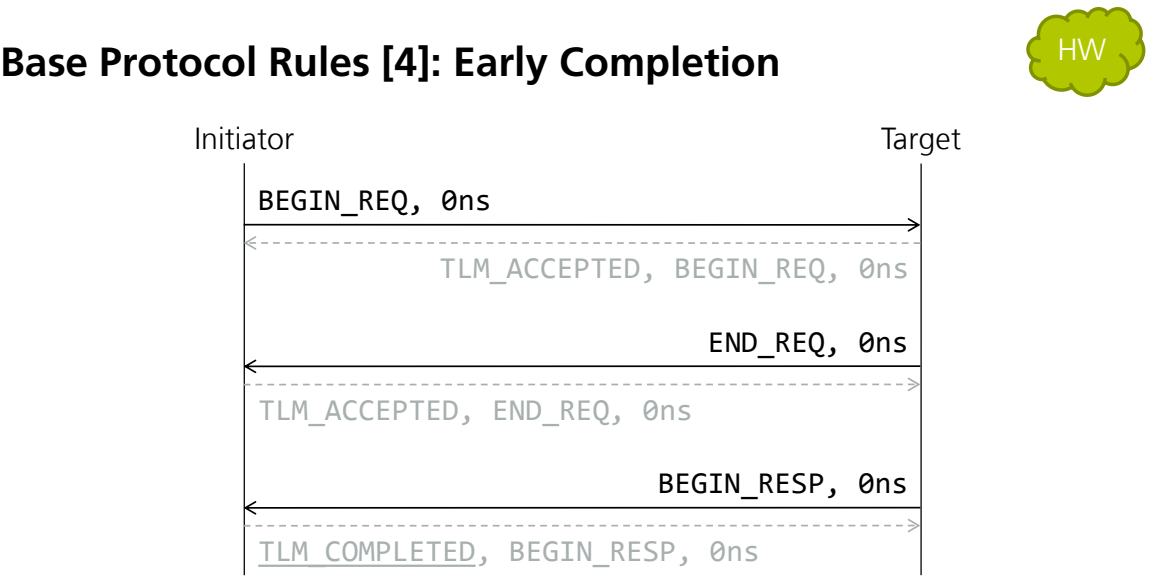


- The initiator sends `BEGIN_REQ` and the target immediately returns `TLM_COMPLETED` (useful for modelling simple I/O, where no ACK is required)
- The targets pre-empts any further communication by signaling a so called *Early completion*
- Initiator should immediacy check the response status of the generic payload object
- With `TLM_COMPLETED` the caller should always ignore the phase argument

Fraunhofer IESE

Your notes:

Base Protocol Rules [4]: Early Completion

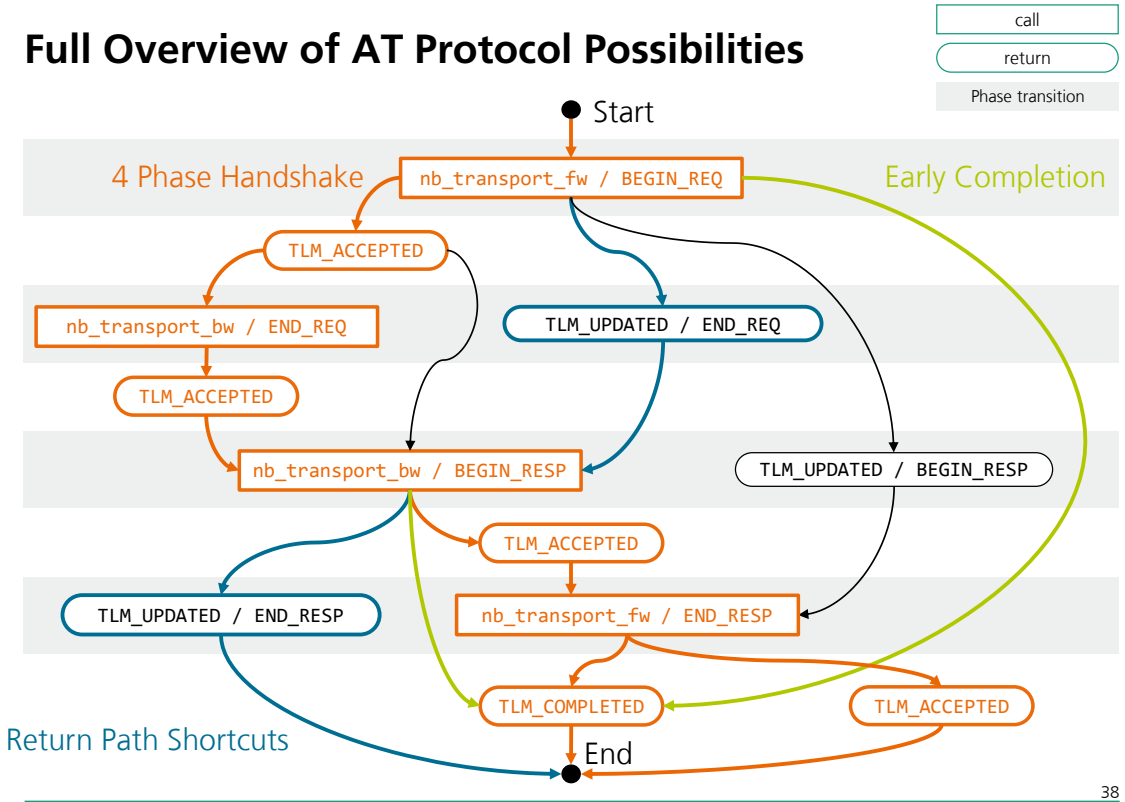


- The initiator can ealy complete the transaction by returning `TLM_COMPLETED`
- The initiator pre-empts therefore any further communication

Fraunhofer IESE

Your notes:

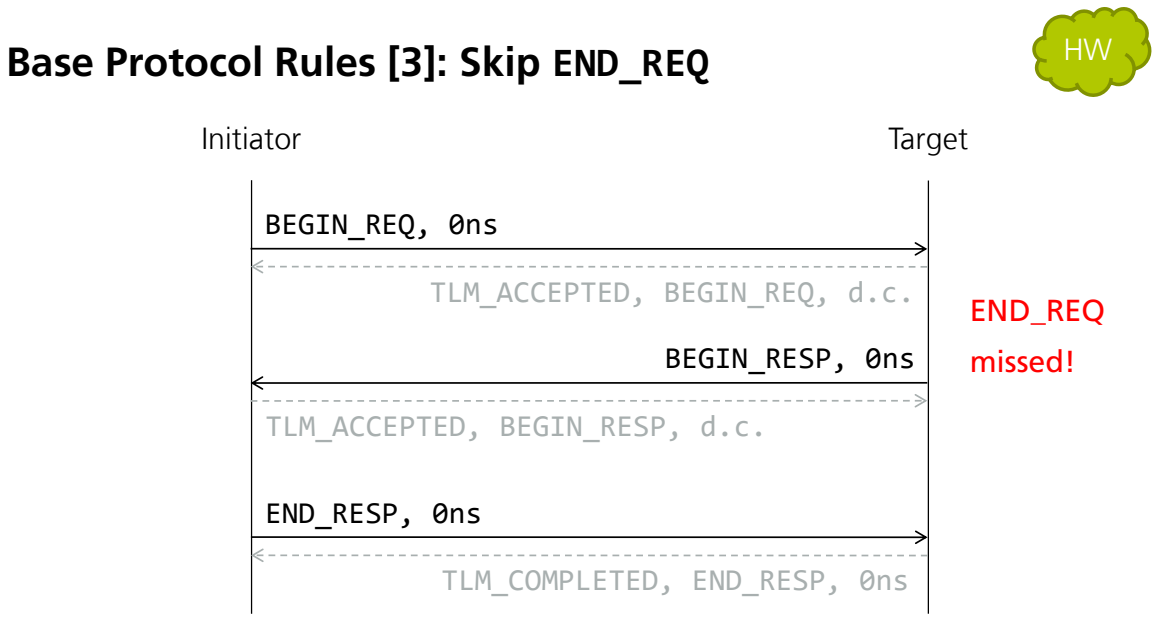
Full Overview of AT Protocol Possibilities



Fraunhofer IESE

Your notes:

Base Protocol Rules [3]: Skip `END_REQ`

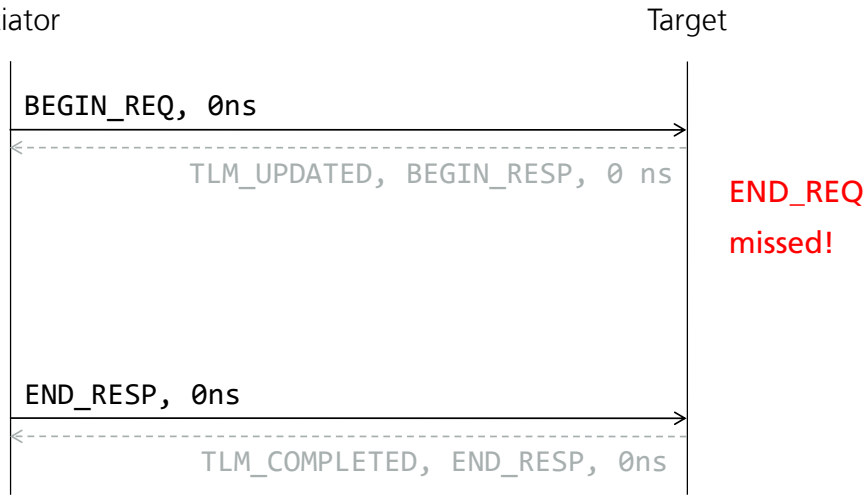


- `END_REQ` is pre-empted by the target sending `BEGIN_RESP` in the next BW call

Fraunhofer IESE

Your notes:

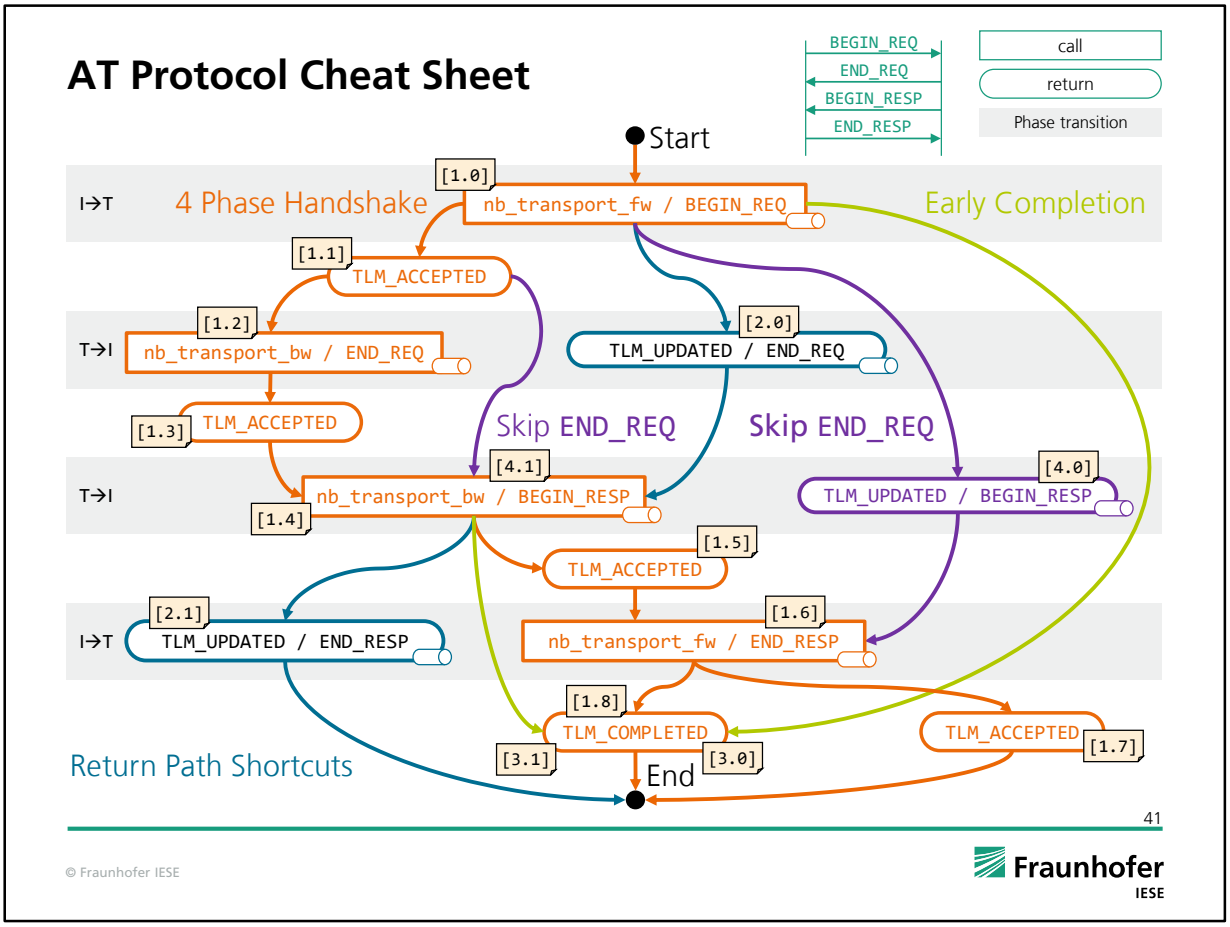
Base Protocol Rules (7): Skip END_REQ (Shortcut)



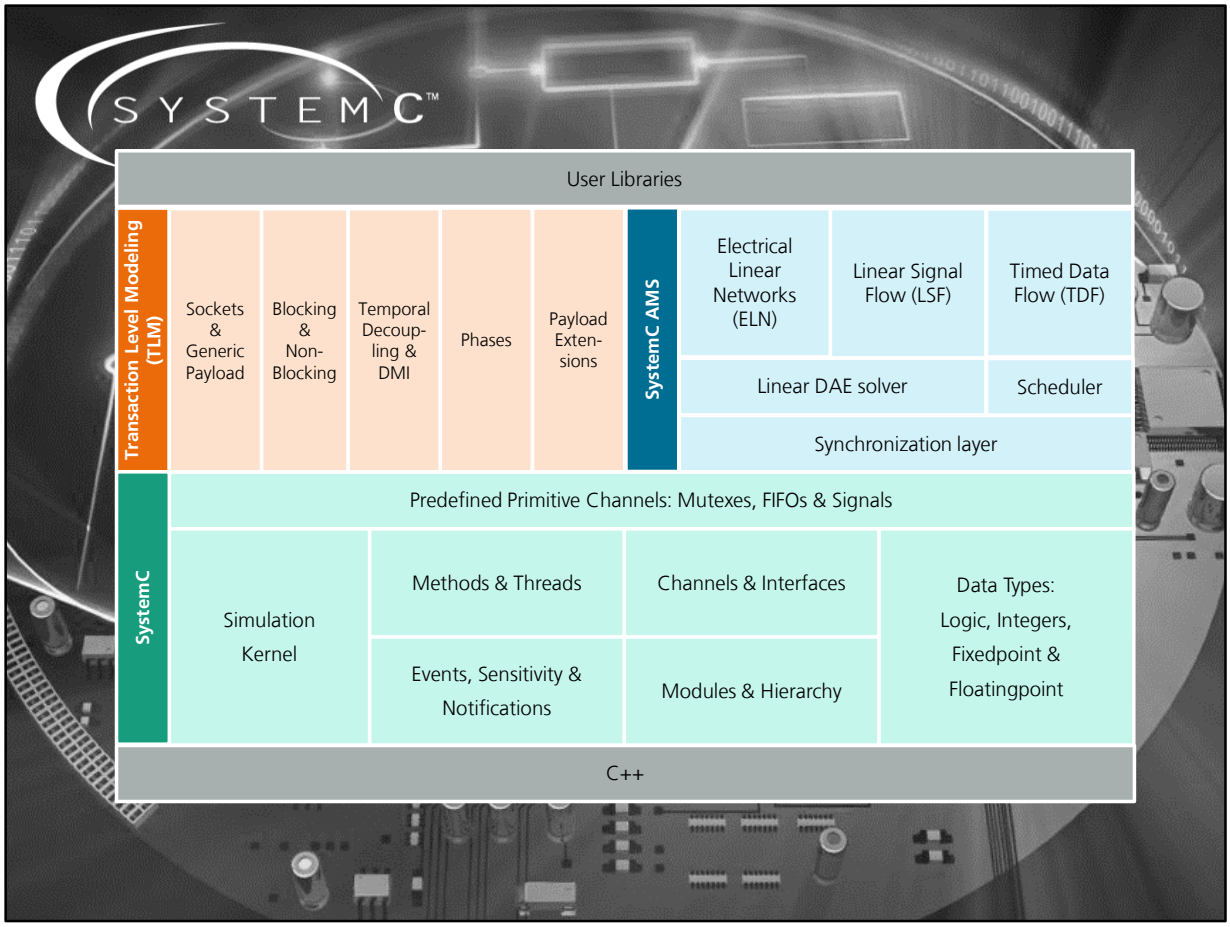
- END_REQ is pre-empted by the target sending directly BEGIN_RESP via TLM_UPDATED over the return path

Fraunhofer IESE

Your notes:


AT Protocol Cheat Sheet
BEGIN_REQ
END_REQ
BEGIN_RESP
END_RESP
call
return
Phase transition
Start
I→T
4 Phase Handshake
[1.0]
nb_transport_fw / BEGIN_REQ
Early Completion
[1.1]
TLM_ACCEPTED
[1.2]
T→I
nb_transport_bw / END_REQ
[2.0]
TLM_UPDATED / END_REQ
[1.3]
TLM_ACCEPTED
Skip END_REQ
Skip END_REQ
[4.1]
T→I
nb_transport_bw / BEGIN_RESP
[1.4]
[4.0]
TLM_UPDATED / BEGIN_RESP
[1.5]
TLM_ACCEPTED
[2.1]
I→T
TLM_UPDATED / END_RESP
[1.6]
nb_transport_fw / END_RESP
[1.8]
TLM_COMPLETED
TLM_ACCEPTED
[1.7]
[3.1]
[3.0]
End
Return Path Shortcuts
41
© Fraunhofer IESE
Fraunhofer
IESE

Your notes:


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++

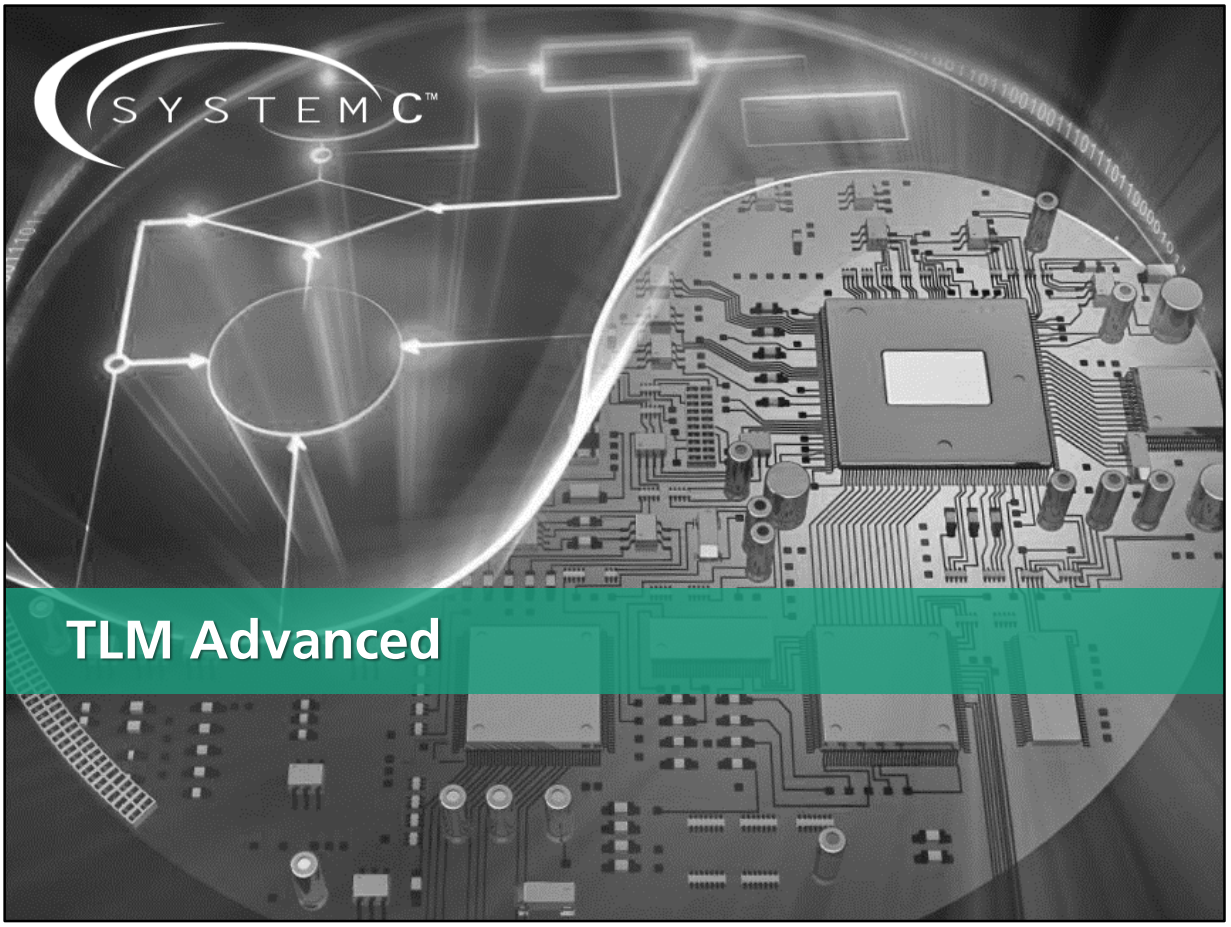
Your notes:

SystemC and Virtual Prototyping

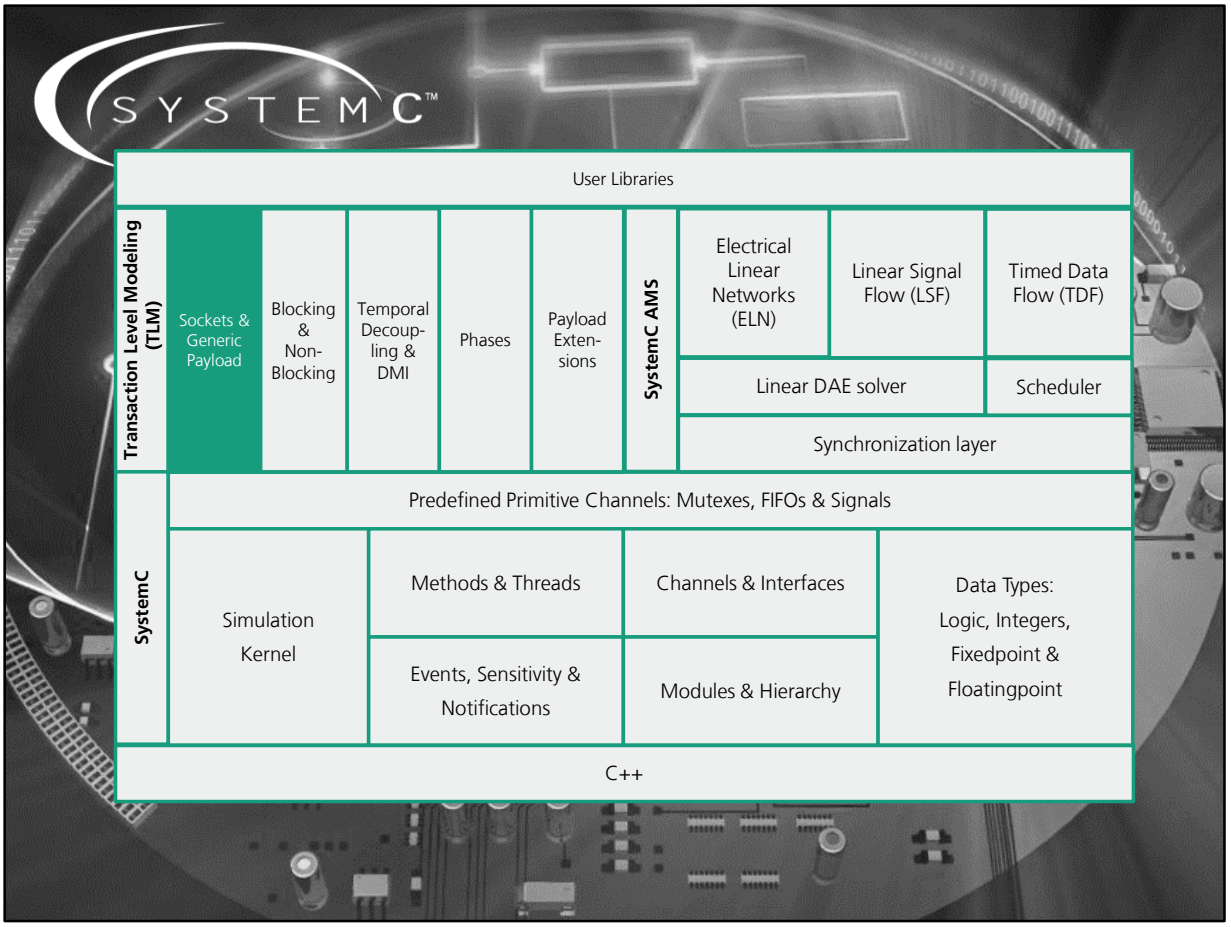
Dr. Matthias Jung, Fraunhofer Institute IESE
matthias.jung@iese.fraunhofer.de

Fraunhofer IESE

Your notes:



Your notes:


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++

Your notes:

Simple Sockets

- "Simple" because they are easy to use – less to code …
- Do not bind sockets to objects, instead register methods with each socket
- Dummy method implementations are provided:
  - `get_direct_mem_ptr`
  - `transport_dbg`
  - `invalidate_direct_mem_ptr`
  - `nb_transport_bw`
- Target need only register either `b_transport` or `nb_transport_fw`
- Automatic conversion between blocking and non-blocking

Fraunhofer IESE

Your notes:

Simple Sockets: Initiator Example

```
#include "tlm_utils/simple_initiator_socket.h"
...

SC_MODULE(Initiator) {
    public:
    tlm_utils::simple_initiator_socket<Initiator> iSocket;
    ...
    public:
    SC_CTOR(Initiator): iSocket("iSocket"), requestInProgress(0), peq(this, &Initiator::peqCallback)
    {
        iSocket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);

        SC_THREAD(process);
        ...
    }

    void process() {
        ...
    }

    tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload& …, tlm::tlm_phase& …, sc_time& …) {
       ...
    }
    ...
};
```

Instantiate simple socket

Register function

Implementation of Function

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_simple_sockets/

Fraunhofer IESE

Your notes:

Simple Sockets: Target Example

```
#include "tlm_utils/simple_target_socket.h"
...
SC_MODULE(Target)
{
    tlm_utils::simple_target_socket<Target> tSocket;
    ...
    SC_CTOR(Target) : tSocket("tSocket"), ...
    {
        tSocket.register_b_transport(this, &Target::b_transport);
        tSocket.register_nb_transport_fw(this, &Target::nb_transport_fw);
    }

    void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay) {
        ...
    }

    tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload& …, tlm::tlm_phase& …, sc_time& …) {
        ...
    }
...
};
```

Instantiate simple target socket

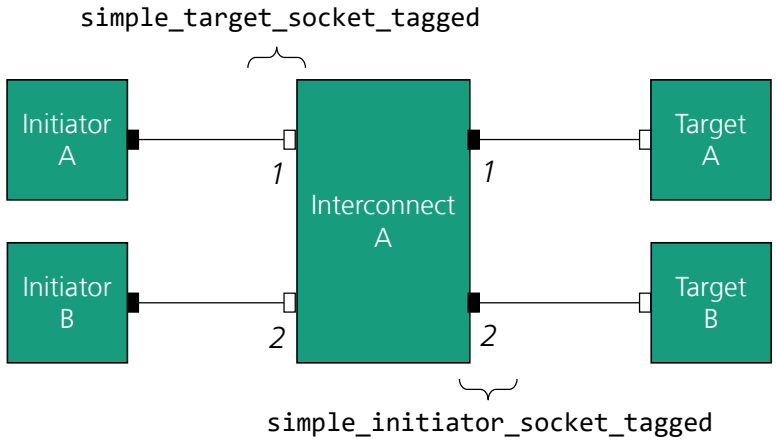
Register functions

Implementation of Functions

Fraunhofer IESE

Your notes:

Tagged Simple Sockets: Interconnect Example



- Register the same callback method with several sockets
- Distinguish origin of incoming transactions using socket id
- Interconnect is usually template class for number of target and initiator sockets

Fraunhofer IESE

Your notes:

Tagged Simple Sockets: Interconnect Example

```
template<unsigned int I, unsigned int T>
SC_MODULE(BUS)
{
    public:
    tlm_utils::simple_target_socket_tagged<BUS> tSocket[T];
    tlm_utils::simple_initiator_socket_tagged<BUS> iSocket[I];

    SC_CTOR(BUS)
    {
        for(unsigned int i = 0; i < T; i++)
        {
            tSocket[i].register_b_transport(this,
                &BUS::b_transport, i);
            tSocket[i].register_nb_transport_fw(this,
                &BUS::nb_transport_fw, i);
        }

        for(unsigned int i = 0; i < I; i++)
        {
            iSocket[i].register_nb_transport_bw(this,
                &BUS::nb_transport_bw, i);
        }
    }

    private:
    std::map<tlm::tlm_generic_payload*, int> bwRoutingTable;
    std::map<tlm::tlm_generic_payload*, int> fwRoutingTable;

 virtual void b_transport( int id,
                           tlm::tlm_generic_payload& trans,
                           sc_time& delay )
    {
        sc_assert(id < T);
        int outPort = routeFW(id, trans, false);
        iSocket[outPort]->b_transport(trans, delay);
    }
```

```
    virtual tlm::tlm_sync_enum nb_transport_fw( int id,
                tlm::tlm_generic_payload& trans,
                tlm::tlm_phase& phase,
                sc_time& delay )
    {
        sc_assert(id < T);
        int outPort = 0;

        if(phase == tlm::BEGIN_REQ) {
            trans.acquire();
            outPort = routeFW(id, trans, true);
        }
        else if(phase == tlm::END_RESP) {
            outPort = fwRoutingTable[&trans];
            trans.release();
        }
        else {
            SC_REPORT_FATAL(name(),"ERROR!");
        }

        return iSocket[outPort]->nb_transport_fw(
                                        trans, phase, delay);
    }

    virtual tlm::tlm_sync_enum nb_transport_bw( int id,
                tlm::tlm_generic_payload& trans,
                tlm::tlm_phase& phase,
                sc_time& delay )
    {
        int inPort = bwRoutingTable[&trans];

        return tSocket[inPort]->nb_transport_bw(
                                        trans, phase, delay);
    }
};
```


Fraunhofer IESE

Your notes:

Simple Sockets: Target Example

```
int routeFW(int inPort, tlm::tlm_generic_payload &trans, bool store)
{
    int outPort = 0;

    if(trans.get_address() < 512)
    {
        outPort = 0;
    }
    else if(trans.get_address() >= 512 && trans.get_address() < 1024)
    {
        trans.set_address(trans.get_address() - 512);
        outPort = 1;
    }
    else {
        trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
    }

    if(store) {
        bwRoutingTable[&trans] = inPort;
       fwRoutingTable[&trans] = outPort;
    }

    return outPort;
}
```

Implementation of the memory map

Correct address, i.e. subtract offset such that target gets always addresses starting at 0

Store from where transaction comes and where it goes

Fraunhofer IESE

Your notes:

Simple Sockets: Target Example

```
int sc_main (...)
{

    Initiator * cpu1   = new Initiator("C1");
    Initiator * cpu2   = new Initiator("C2");

    Target * memory1   = new Target("M1");
    Target * memory2   = new Target("M2");

    Interconnect<2,2> * bus = new BUS<2,2>("B1");

    cpu1->iSocket.bind(bus->tSocket[0]);
    cpu2->iSocket.bind(bus->tSocket[1]);

    bus->iSocket[0].bind(memory1->tSocket);
    bus->iSocket[1].bind(memory2->tSocket);

    sc_start();

    return 0;
}
```

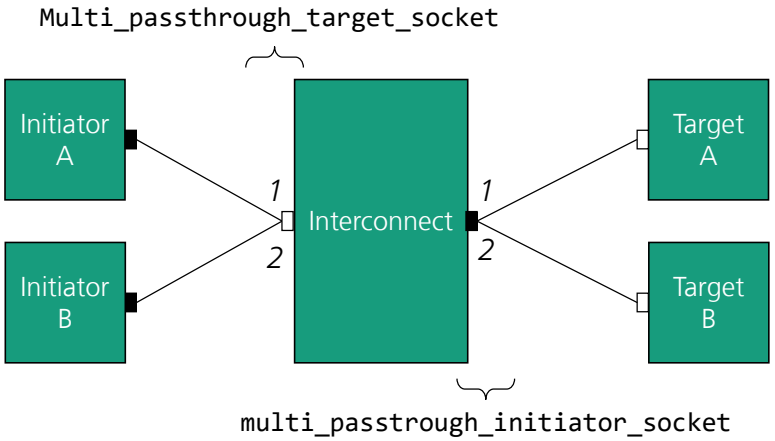
Number of components must be known at compile time!

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_simple_sockets/

Fraunhofer IESE

Your notes:

Multipasstrough Sockets



- Similar to Tagged sockets: also an id is used for identification
- The id is determined by the order of the binding to the multipasstrough socket
- Dynamic binding, number of comp. does not have to be known at compile time

Fraunhofer IESE

Your notes:

Multipasstrough Sockets: Interconnect Example

```
SC_MODULE(BUS)
{
    public:
    tlm_utils::multi_passthrough_target_socket<BUS> tSocket;
    tlm_utils::multi_passthrough_initiator_socket<BUS> iSocket;

    SC_CTOR(Interconnect) : tSocket("tSocket"), iSocket("iSocket")
    {
        tSocket.register_b_transport(this, &BUS::b_transport);
        tSocket.register_nb_transport_fw(this, &BUS::nb_transport_fw);
        iSocket.register_nb_transport_bw(this, &BUS::nb_transport_bw);
    }

    private:
    std::map<tlm::tlm_generic_payload*, int> bwRoutingTable;
    std::map<tlm::tlm_generic_payload*, int> fwRoutingTable;

    void b_transport( int id, tlm::tlm_generic_payload& trans, sc_time& delay ) {
        ...
    }

    tlm::tlm_sync_enum nb_transport_fw( int id,  ... ) {
        ...
    }

    virtual tlm::tlm_sync_enum nb_transport_bw( int id, ...) {
        ...
    }
};
```


Fraunhofer IESE

Your notes:

Multipasstrough Sockets: Interconnect Example

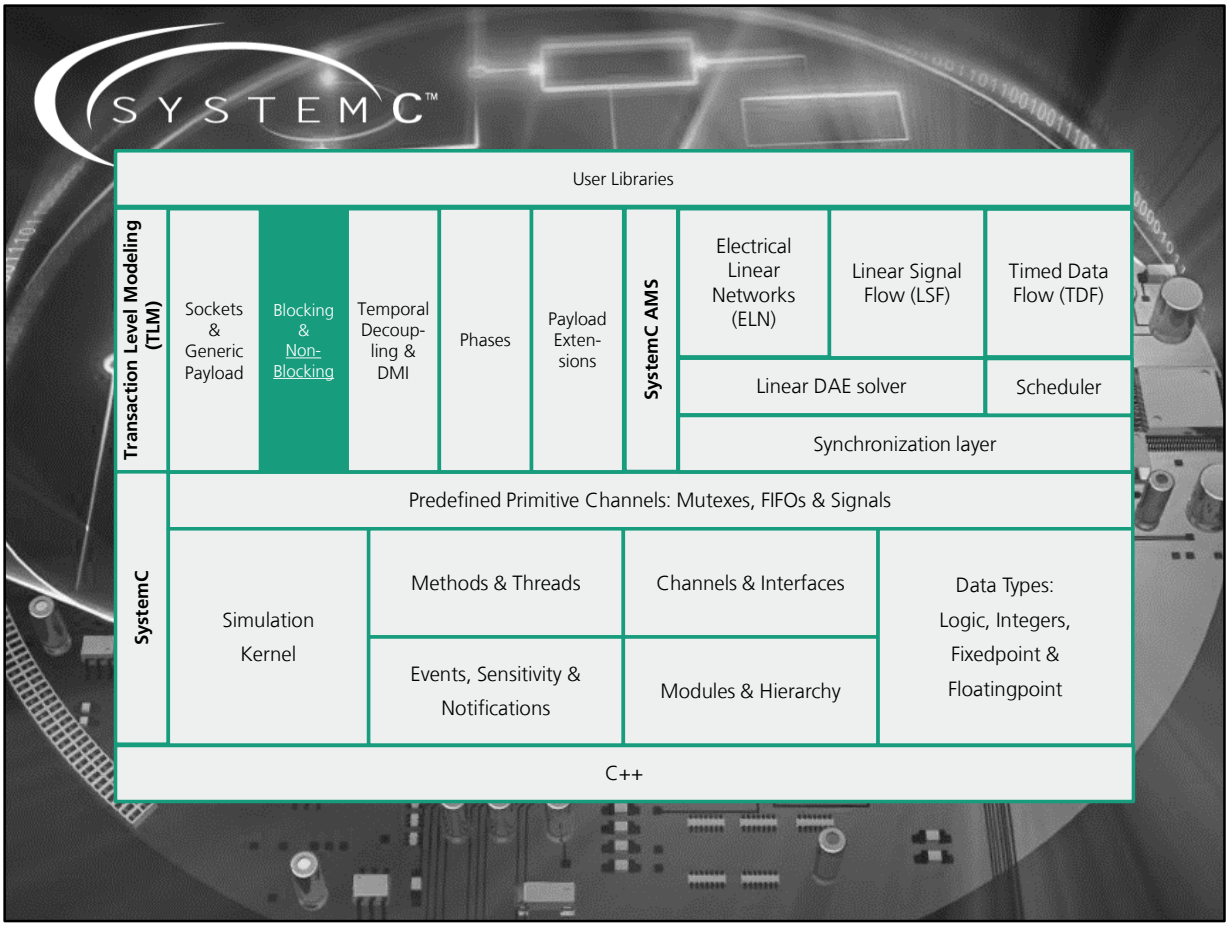
```
int sc_main (...)
{
    Initiator * cpu1   = new Initiator("C1");
    Initiator * cpu2   = new Initiator("C2");

    Target * memory1   = new Target("M1");
    Target * memory2   = new Target("M2");

    Interconnect * bus = new BUS("B1");

    cpu1->iSocket.bind(bus->tSocket);
    cpu2->iSocket.bind(bus->tSocket);

    bus->iSocket.bind(memory1->tSocket);
    bus->iSocket.bind(memory2->tSocket);

    sc_start();

    return 0;
}
```

The order of the binding determines the ids for the function calls!

Try code on github:
https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_multipasstrough_sockets

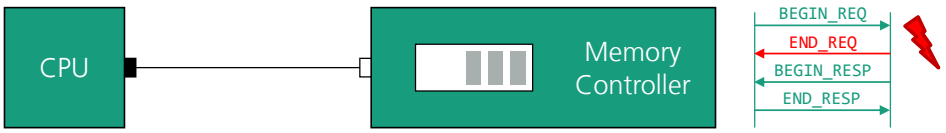
Fraunhofer IESE

Your notes:


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++

Your notes:

Modelling of Backpressure



- The exclusion rules enable flow control like back pressure:
  - E.g. if an input buffer of a target is full the target can defer the sending of `END_REQ` for the transaction that filled the buffer, until the buffer has available space again
  - An RTL ready signal can be modeled by deferring `END_REQ`
  - However, since it is non-blocking, the target can do something else, e.g. sending
- Also Response exclusion rule must be honored!

Fraunhofer IESE

Your notes:

Modelling of Backpressure

HW

```
DECLARE_EXTENDED_PHASE(INTERNAL);

SC_MODULE(Target) {
    bool responseInProgress;
    tlm::tlm_generic_payload* endRequestPending;
    tlm_utils::peq_with_cb_and_phase<Target> peq;
    unsigned int numberOfTransactions;
    unsigned int bufferSize;
    std::queue<tlm::tlm_generic_payload*> responseQueue;
    ...

    void peqCallback(...) {
        sc_time delay;

        if(phase == tlm::BEGIN_REQ) {
            trans.acquire();
            if (numberOfTransactions < bufferSize) {
                sendEndRequest(trans);
            } else {
                endRequestPending = &trans;
            }
        } else if (phase == tlm::END_RESP) {
            ...
            numberOfTransactions--;
            ...
            if (responseQueue.size() > 0) {
               gp * next = responseQueue.front();
               responseQueue.pop();
               sendResponse(*next);
            }
            if (endRequestPending) {
               sendEndRequest(*endRequestPending);
               endRequestPending = 0;
            }
        }
        else if(phase == INTERNAL) {
            executeTransaction(trans);
            if (responseInProgress) {
                responseQueue.push(&trans);
            } else {
                sendResponse(trans);
            }
        }
    }
```

Internal protocol phase, instead of using a sensitive process!

Check if input buffer is full

Reduce transaction counter

Send next response

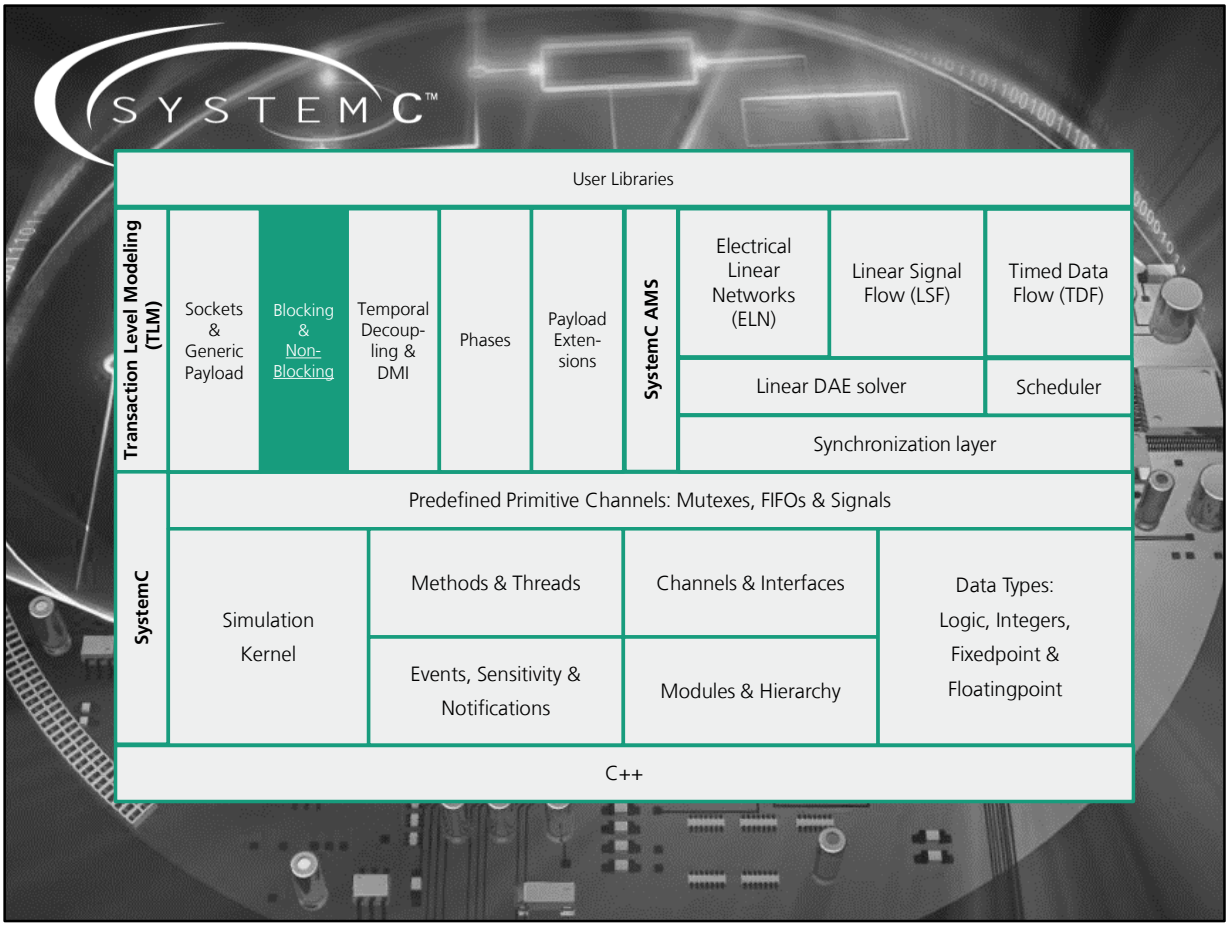
Unblock Initiator

Honor response exclusion rule

```
    void sendEndRequest(tlm::tlm_generic_payload& trans) {
        tlm::tlm_phase bw_phase;
        sc_time delay;
        bw_phase = tlm::END_REQ;
        delay = randomDelay();

        tlm::tlm_sync_enum status;
        status = socket->nb_transport_bw(trans, bw_phase, delay);
        delay = delay + randomDelay();
        peq.notify(trans, INTERNAL, delay);

        numberOfTransactions++;
        printBuffer(bufferSize, numberOfTransactions);
    }

    void sendResponse(tlm::tlm_generic_payload& trans) {
        tlm::tlm_sync_enum status;
        tlm::tlm_phase bw_phase;
        sc_time delay;

        responseInProgress = true;
        bw_phase = tlm::BEGIN_RESP;
        delay = SC_ZERO_TIME;
        status = socket->nb_transport_bw( trans, bw_phase, delay );

        if (status == tlm::TLM_UPDATED) {
            peq.notify(trans, bw_phase, delay);
        }
        else if (status == tlm::TLM_COMPLETED) {
            numberOfTransactions--;
            responseInProgress = false;
        }
        trans.release();
    }
};
```

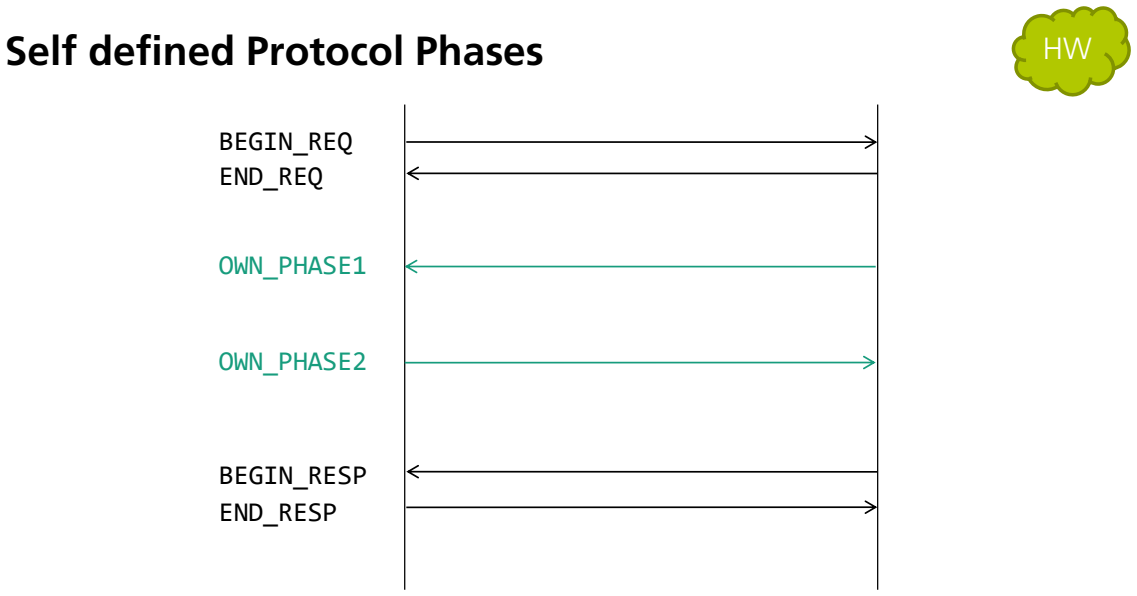
Try code on github: https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_at_backpressure

Your notes:


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++

Your notes:

Self defined Protocol Phases

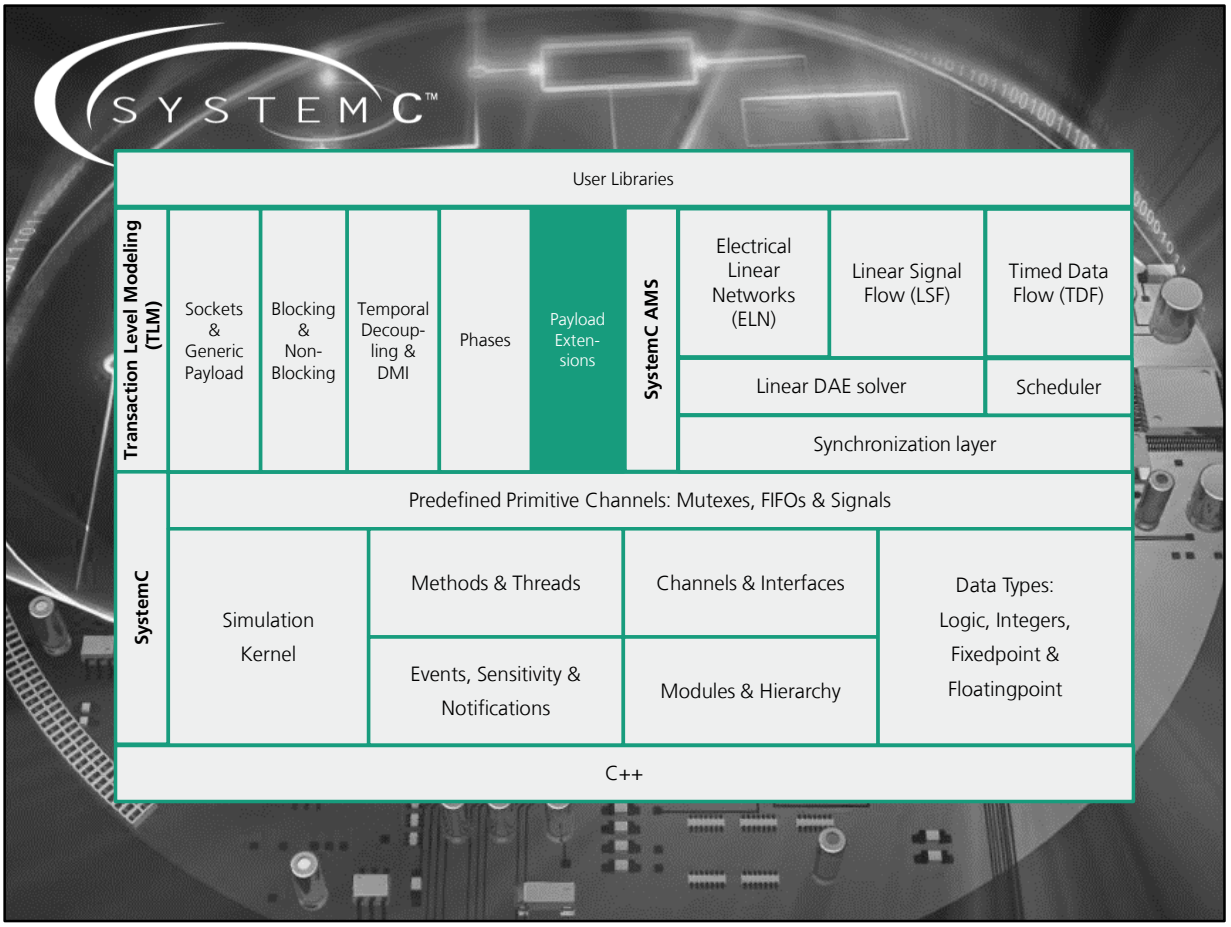


- Base protocol can be enhanced by phase extensions
- Usage of the macro: `DECLARE_EXTENDED_PHASE(INTERNAL);`
- More synchronization points → higher accuracy → slower simulation speed

Interleaving of transaction is possible

Fraunhofer IESE

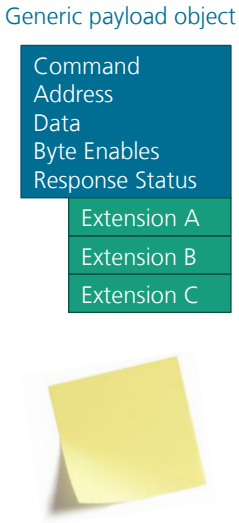
Your notes:


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++

Your notes:

Payload Extensions

- Payload extensions are a flexible and powerful method to carry additional non-standard attributes (e.g. priority, …)
- Extensions are attached to the normal generic payload
- Extensions can be:
  - Ignorable (ensures compatibility with base protocol)
  - Mandatory (a new protocol is needed!)
  - Private (only used by a single module e.g. Interconnect for storing routing information instead using maps)
  - Sticky (remain when transaction is returned to a pool)
  - Auto (freed when transaction is returned to a pool)



Fraunhofer IESE

Your notes:

Payload Extensions Example

```
class routingExtension : public
tlm::tlm_extension<routingExtension> {
    private:
    unsigned int inputPortNumber;
    unsigned int outputPortNumber;

    public:
    routingExtension(unsigned int i, unsigned int o) :
        inputPortNumber(i), outputPortNumber(o)
    {}

    tlm_extension_base* clone() const {
        return new routingExtension(
            inputPortNumber, outputPortNumber);
    }

    void copy_from(const tlm_extension_base& ext) {
        const routingExtension& cpyFrom =
                static_cast<const routingExtension&>(ext);
        inputPortNumber = cpyFrom.getInputPortNumber();
        outputPortNumber = cpyFrom.getOutputPortNumber();
    }

    unsigned int getInputPortNumber() const {
        return inputPortNumber;
    }

    unsigned int getOutputPortNumber() const {
        return outputPortNumber;
    }
};
```

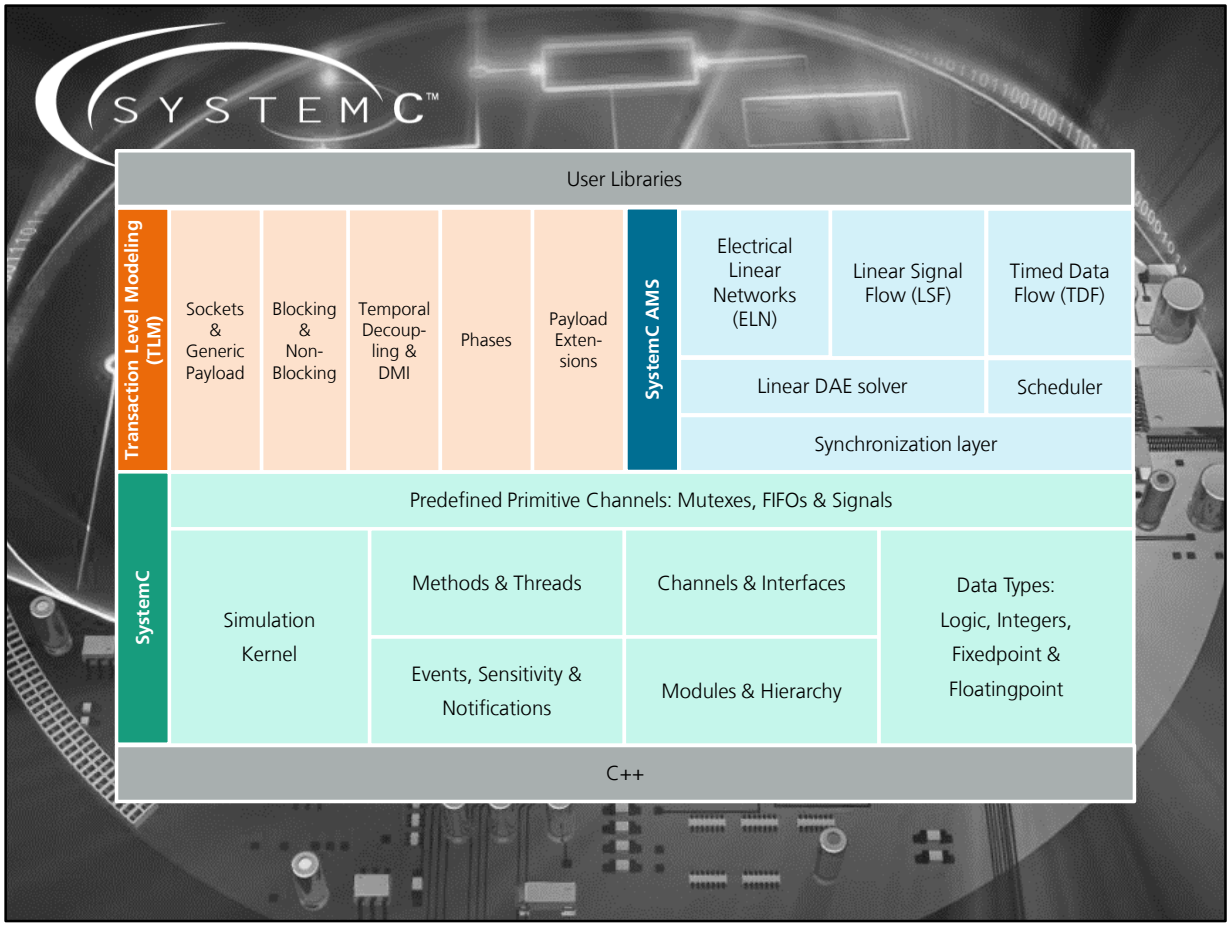
```
SC_MODULE(Interconnect)
{
    ...
    routingExtension* ext;
    ext = new routingExtension(
        inPort, outPort);
    trans.set_auto_extension(ext);
    ...
    routingExtension *ext = NULL;
    trans.get_extension(ext);
    outPort = ext->getOutputPortNumber();
    ...
};
```

Try code on github: https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_payload_extensions/

65

Fraunhofer IESE

Your notes:


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++

Your notes: