SystemC and Virtual Prototyping

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

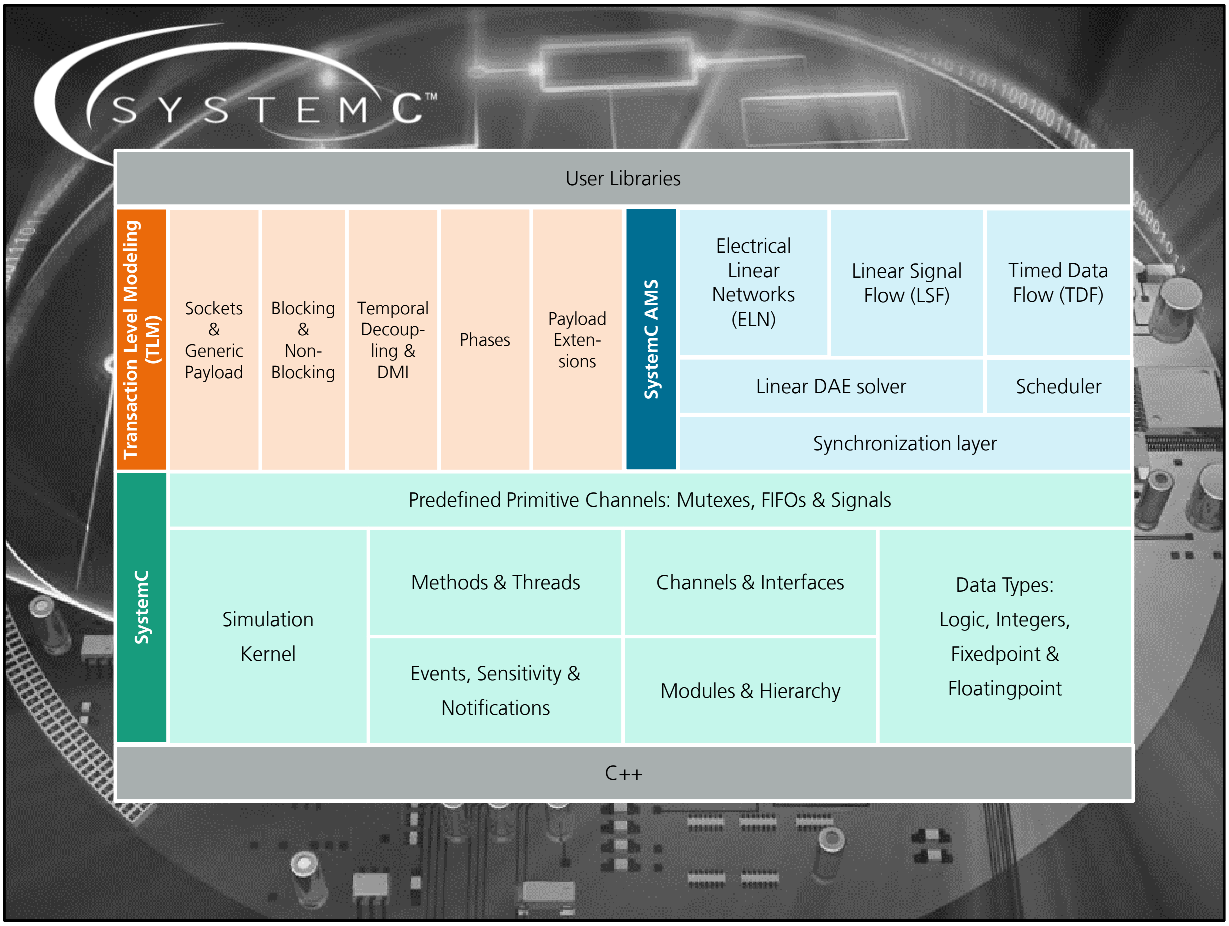
Fraunhofer
IESE

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

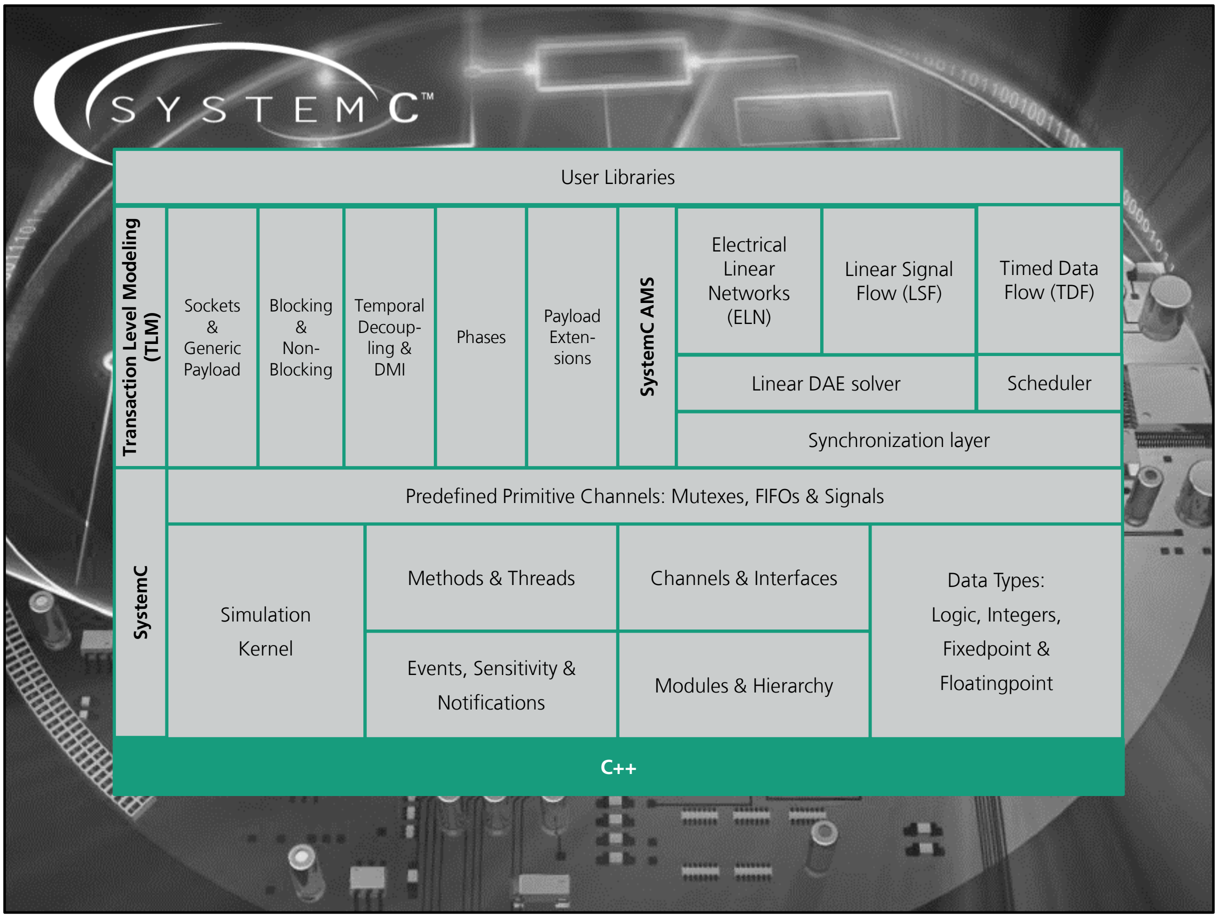

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoupling & DMI
Phases
Payload Extensions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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

Your notes:

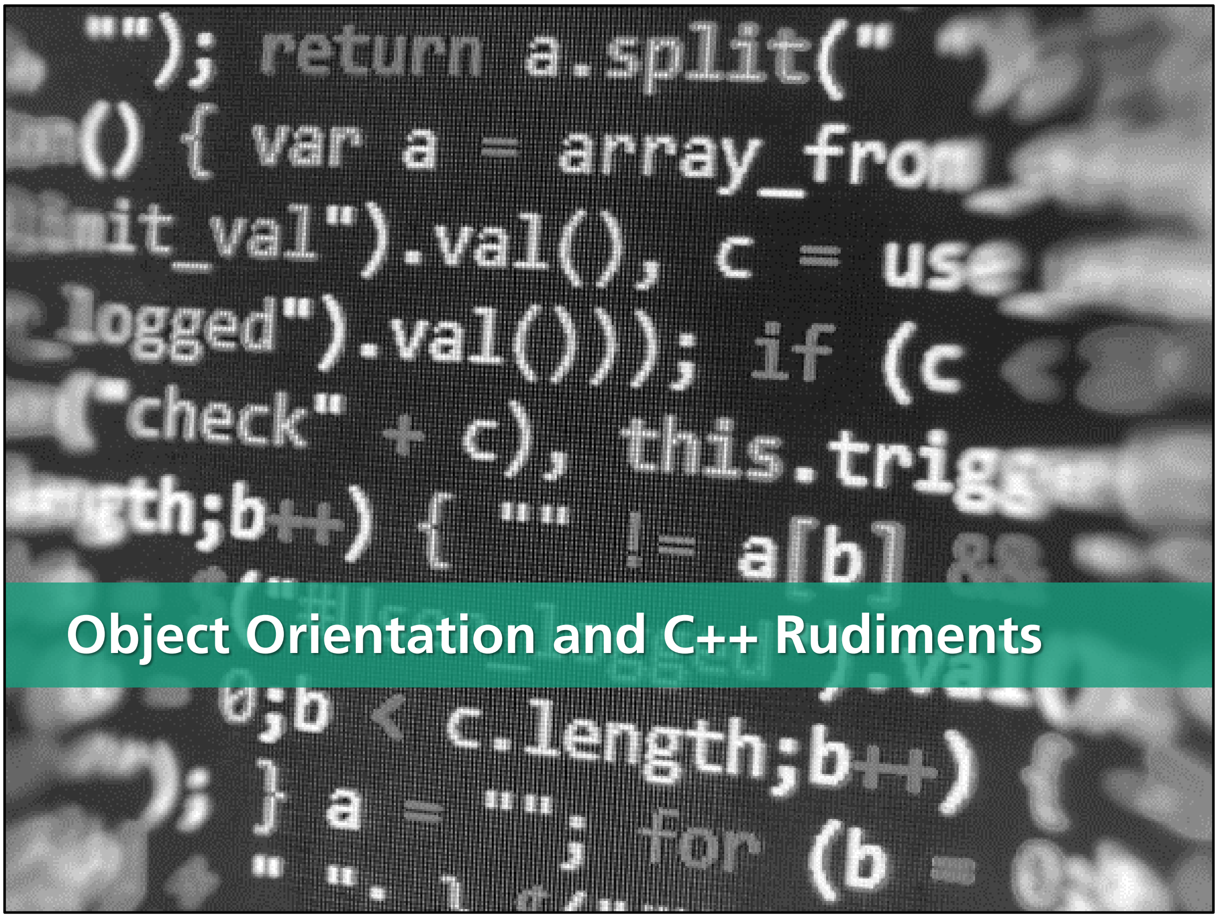
Time Planning

	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

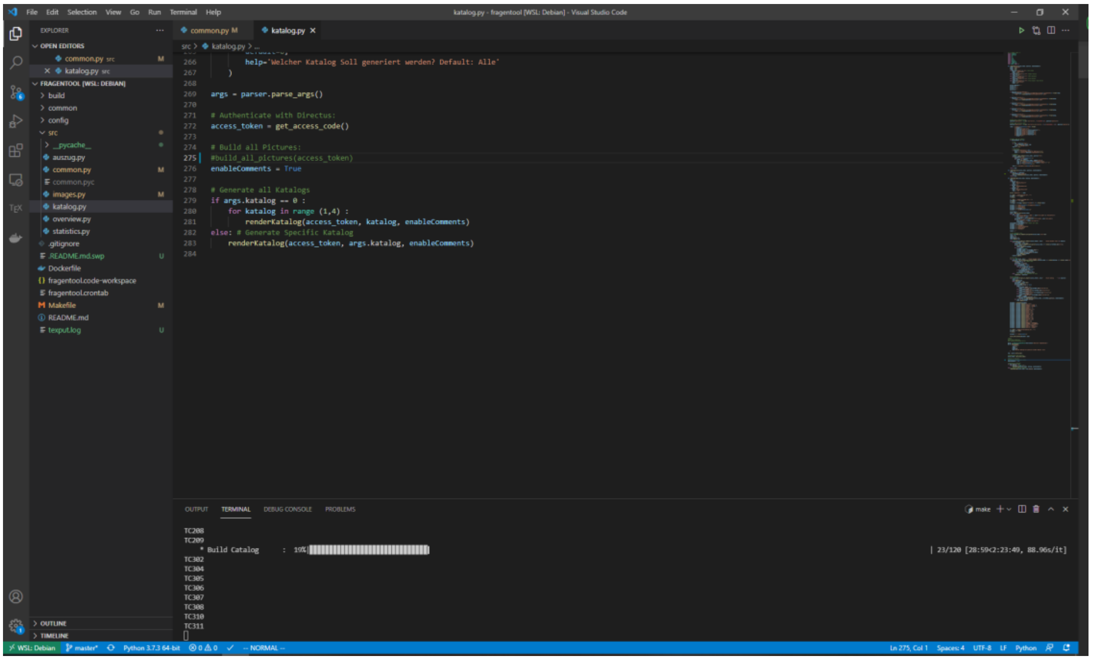

SYSTEMC™
User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer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



7

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

```
#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:

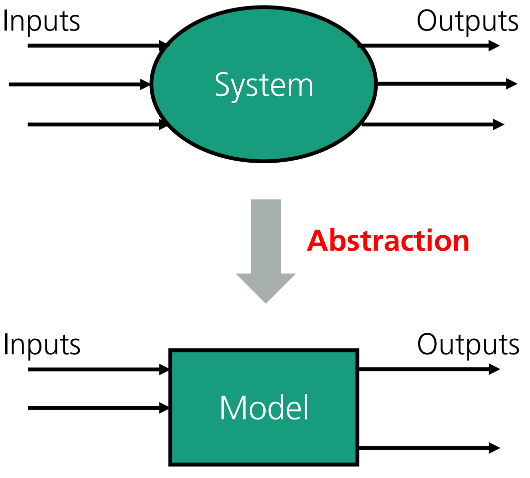
System and Model

- 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

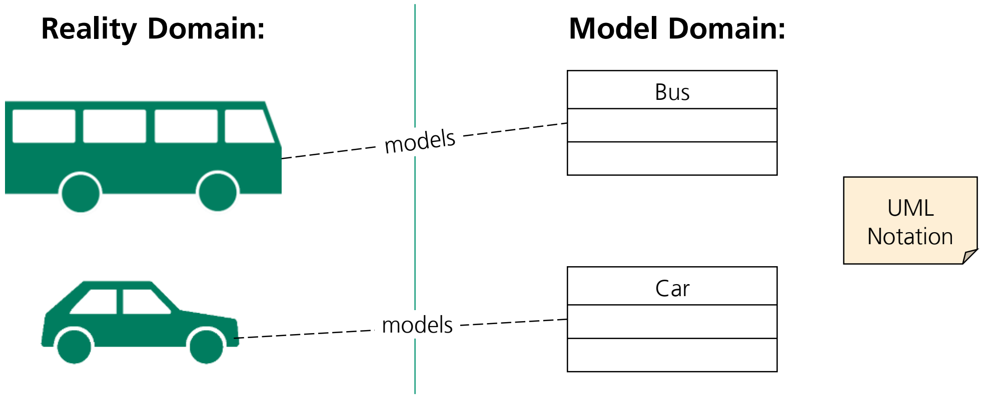


Your notes:

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!



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.

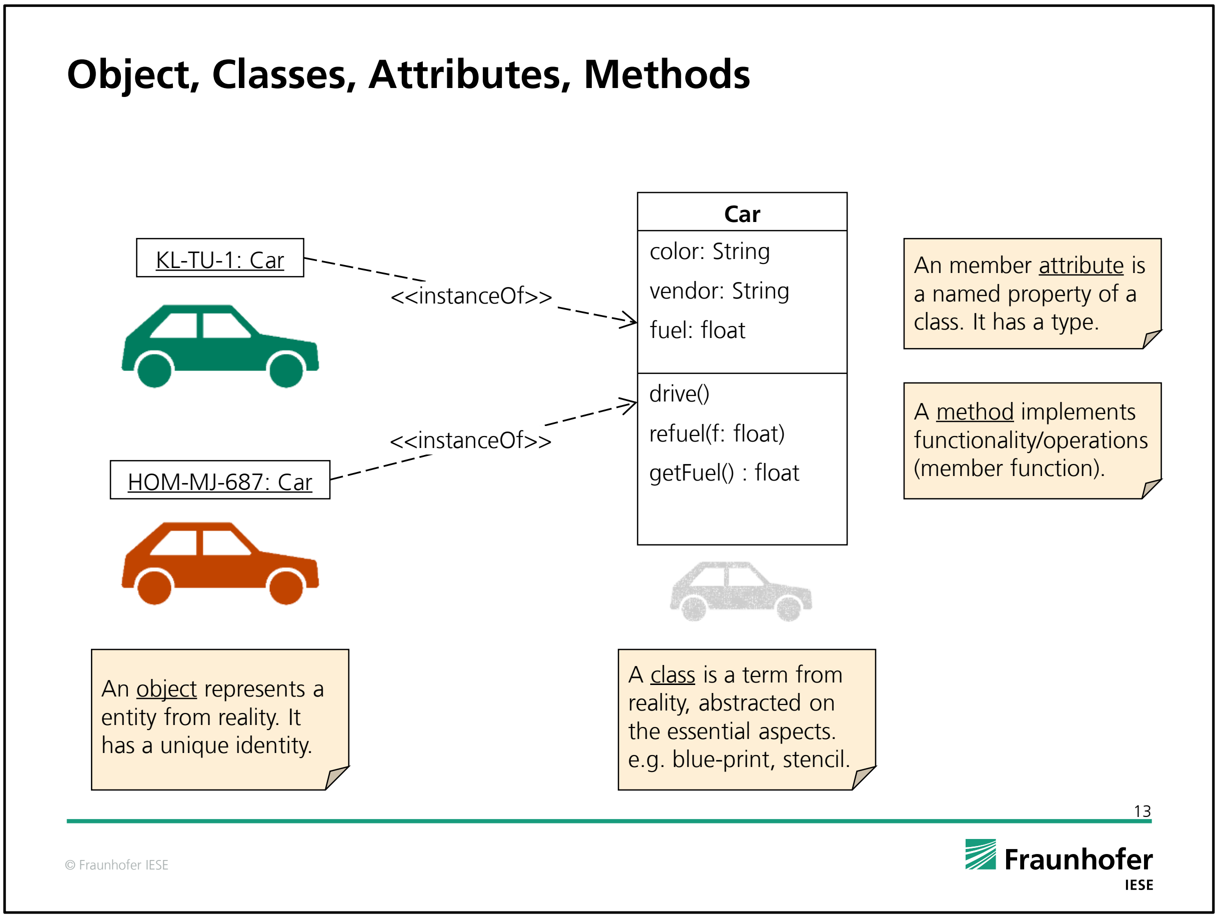
Your notes:

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:



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

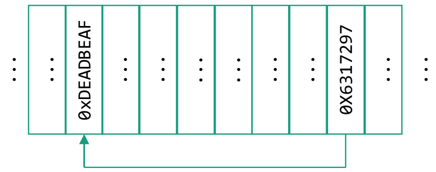
Pointers (->)

Object of class car are created by passing constructor arguments!

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:

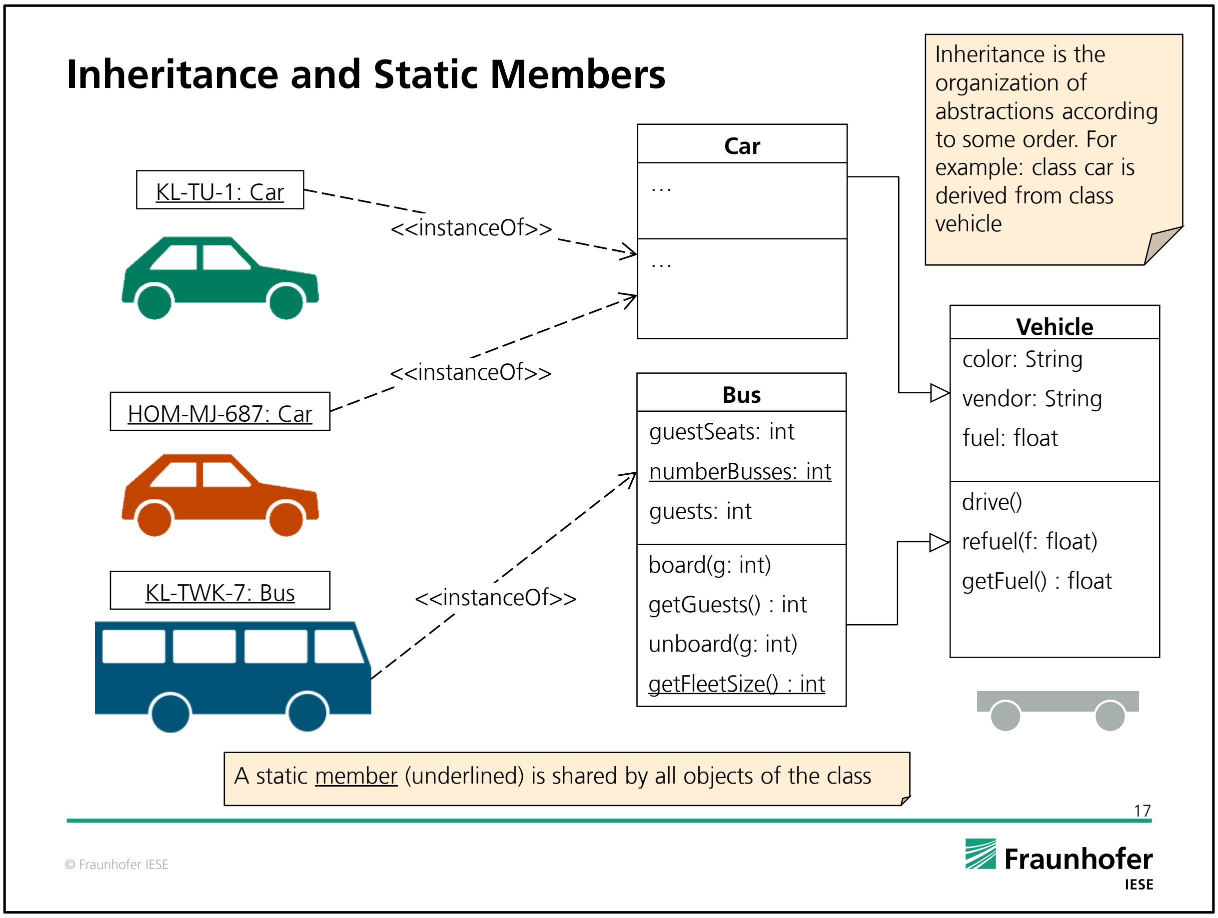
Call by Reference and Call by Value

```
#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 <u>copied</u> 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

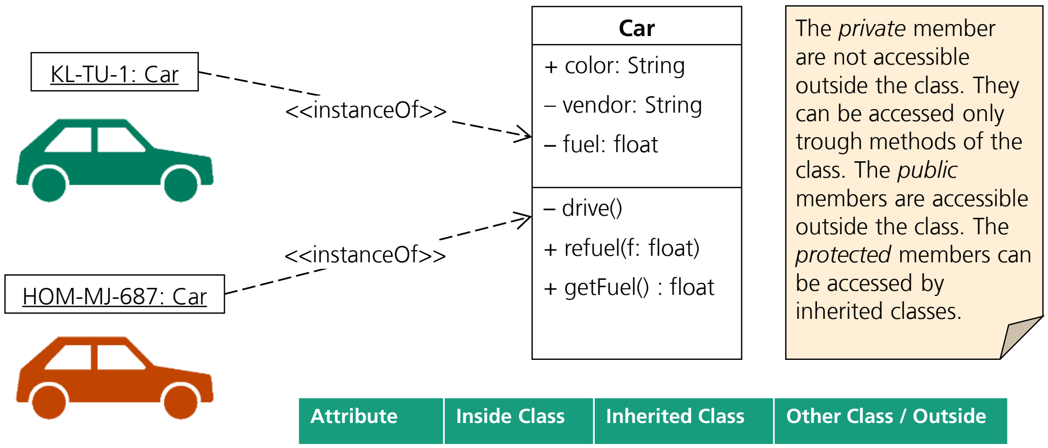
Output will be:
2
2
2

18

Fraunhofer IESE

Your notes:

Encapsulation / Visibility / Information Hiding



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

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

19

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

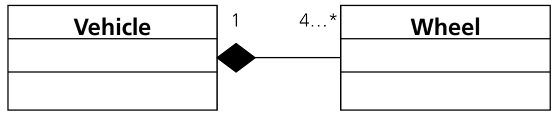
```
class Base {
public:
  Base(): a(0) {}
  virtual ~Base() {}
protected:
  int a;
};

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

```cpp
#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!




```cpp
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

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

22

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

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

```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

- 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)

```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 pointers to abstract classes can be created, no objects!
- Child classes <u>must</u> implement virtual function! Otherwise compiler crashes!
- Why using it? For structuring! For defining <u>Interfaces → Exchangeability during Runtime</u> 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.

26

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.

27

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.

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.

29

Fraunhofer IESE

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

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 …

```
#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;
}
```

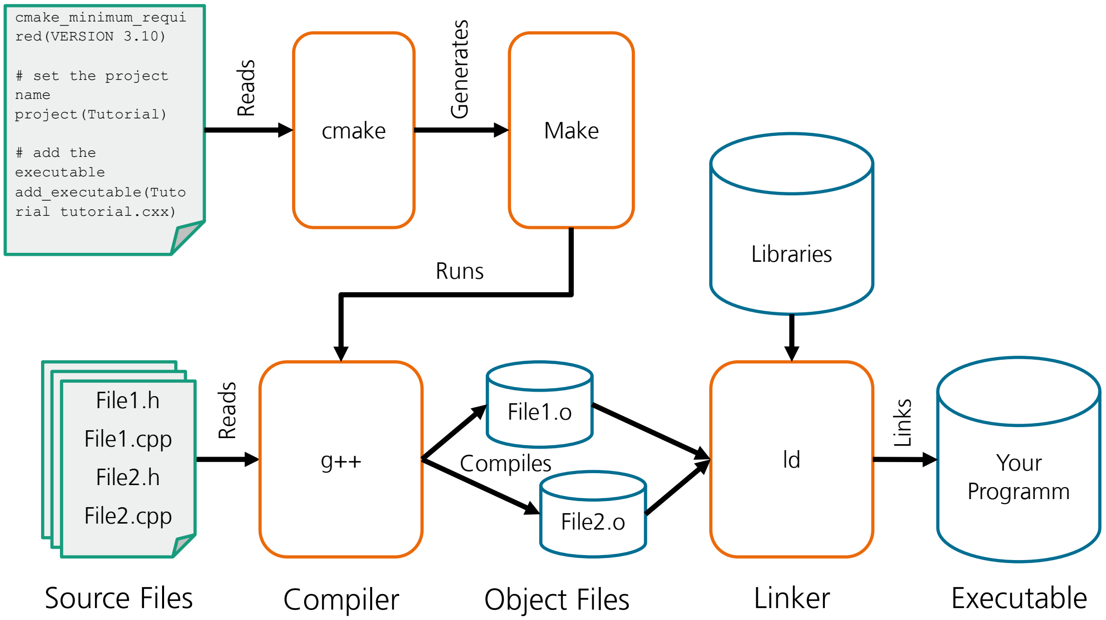
Whats new in C++ 11 …

- 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



34

Fraunhofer IESE

Your notes:

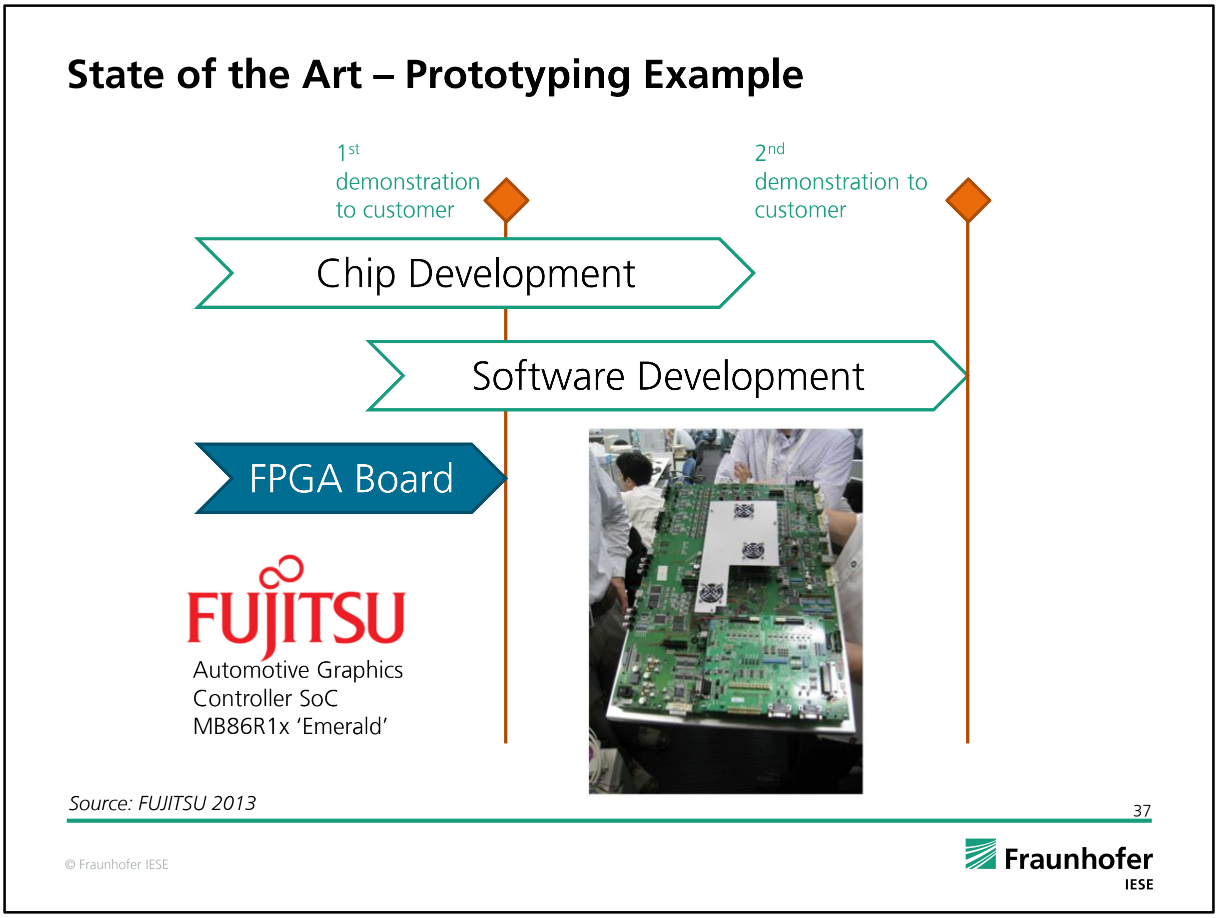
SystemC and Virtual Prototyping

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

Fraunhofer
IESE

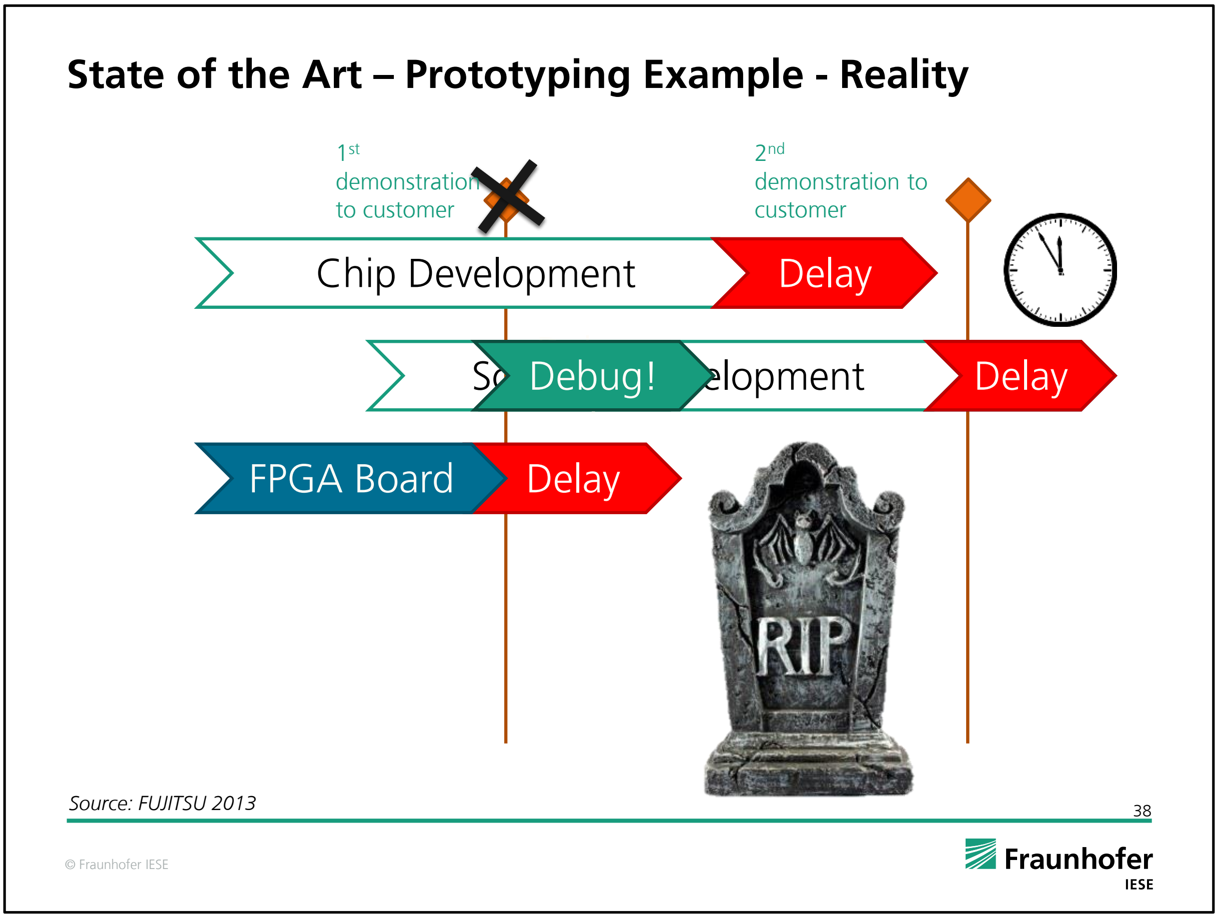
Your notes:

Your notes:



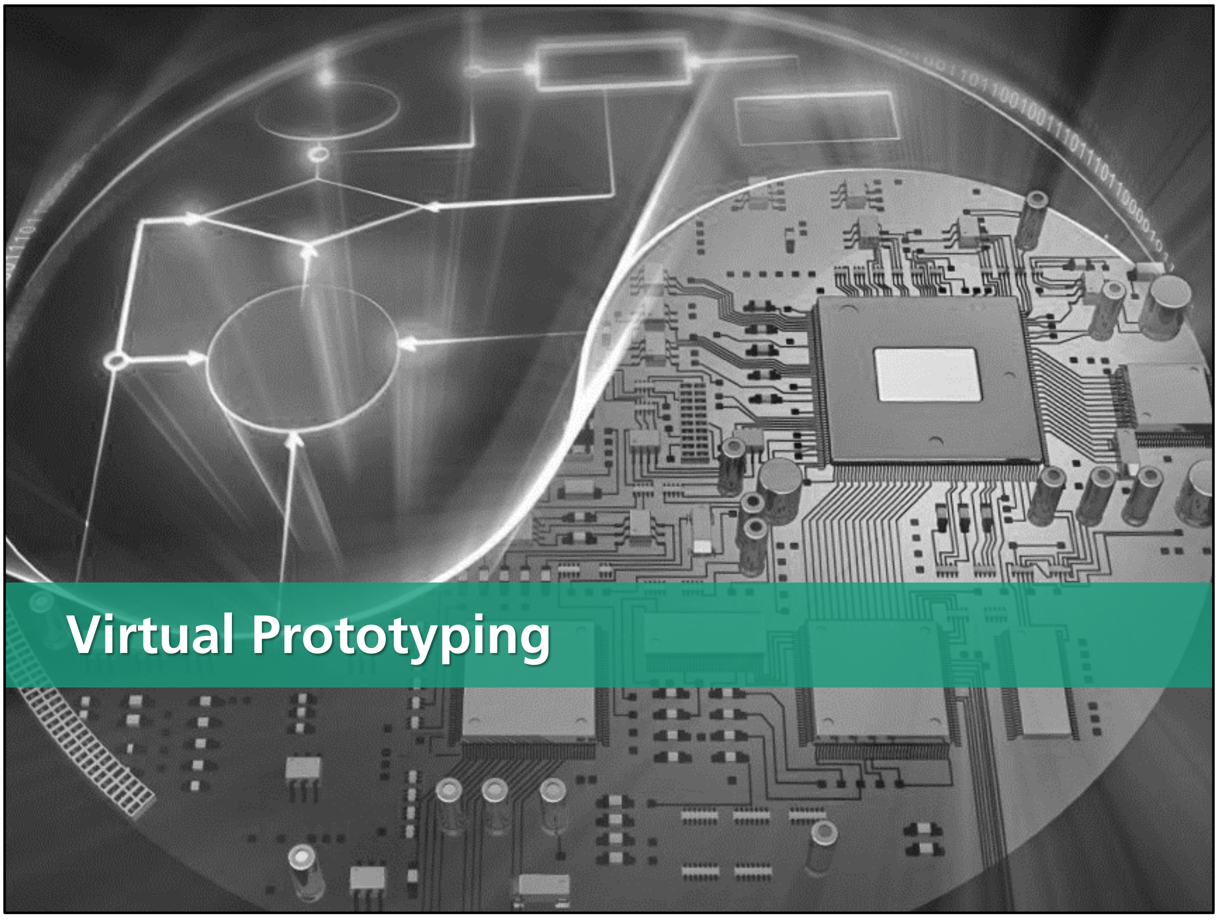
Today's companies have to deal with complex hardware architectures like the before presented multi-core systems. Moreover, they have a constant pressure to deliver their products quickly because of many competitors on the market. The old-established design-flow procedures have a performance problem due 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:



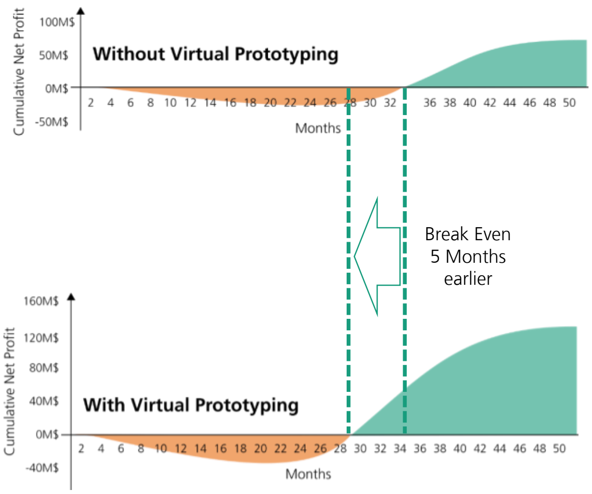
However, FPGA development can be also challenging. In this Example the development of the FPGA board was delayed, such that the final deadline was missed.

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

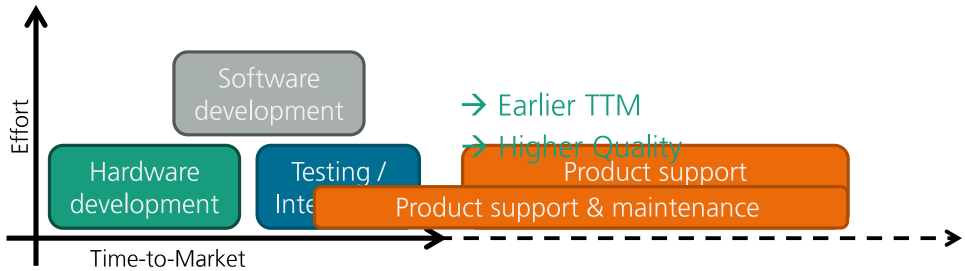
Fraunhofer IESE

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
- <u>Concurrent HW and SW development</u>
- <u>Design Space Exploration (DSE) for HW</u>
- 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?

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:

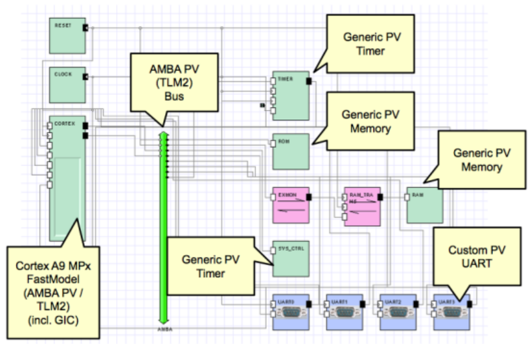
Virtual Prototyping will Help!

2 days after silicon 'Quake' was running

Chip Development

Software Development

Virtual Prototype



FUJITSU

Automotive Graphics Controller SoC MB86R1x 'Emerald'

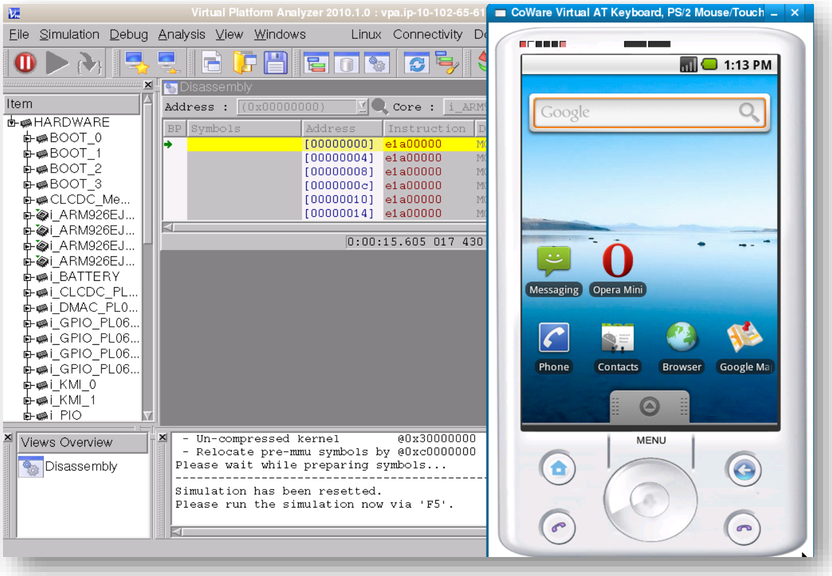
43

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?



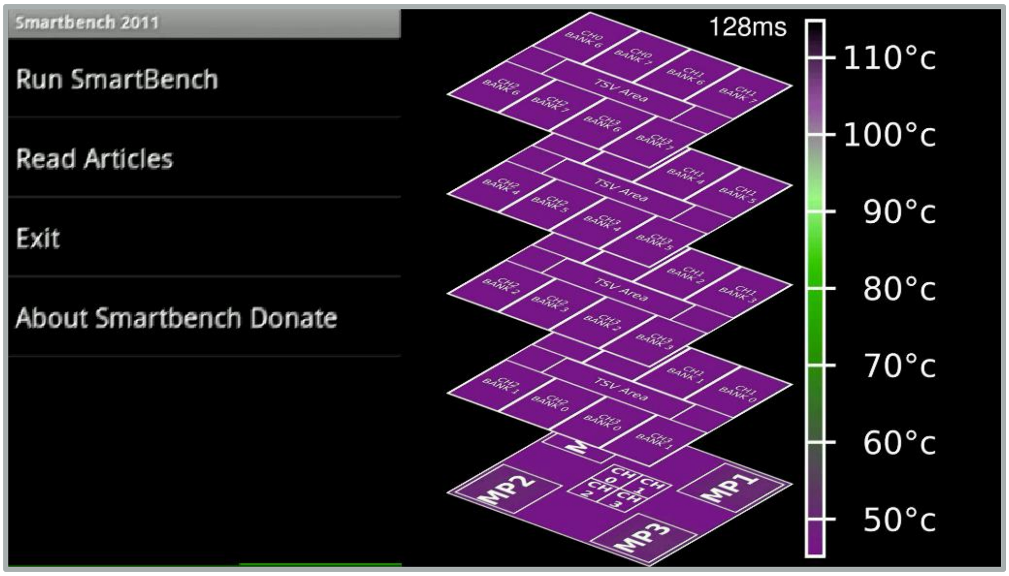
44

© Fraunhofer IESE

Fraunhofer
IESE

Your notes:

Simulate Future Smartphone with 3D Integrated Circuits



Fraunhofer IESE

Your notes:

Accuracy vs. Speed Trade-Off

46

Fraunhofer IESE

Your notes:

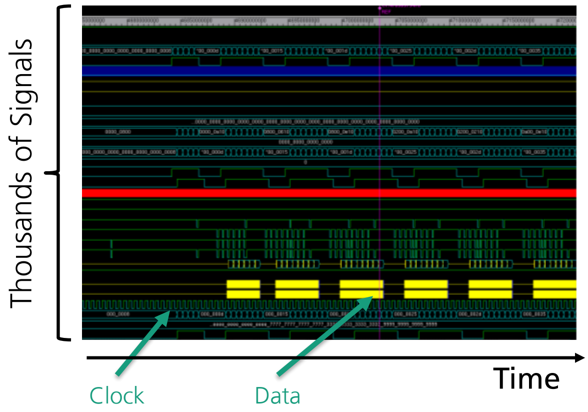
Accuracy vs. Speed Trade-Off

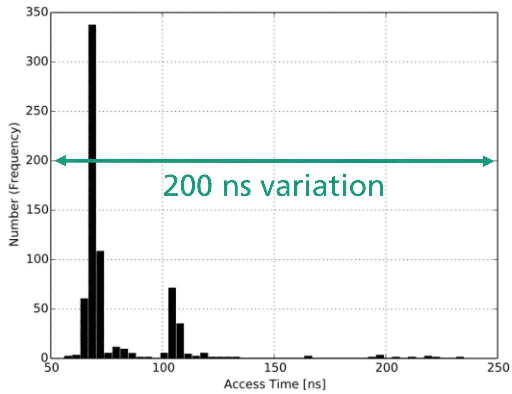
E.g. RTL Simulations:

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

E.g. System Level Simulations:

- Fast
- Large flexibility
- Inaccurate





Fraunhofer IESE

Your notes:

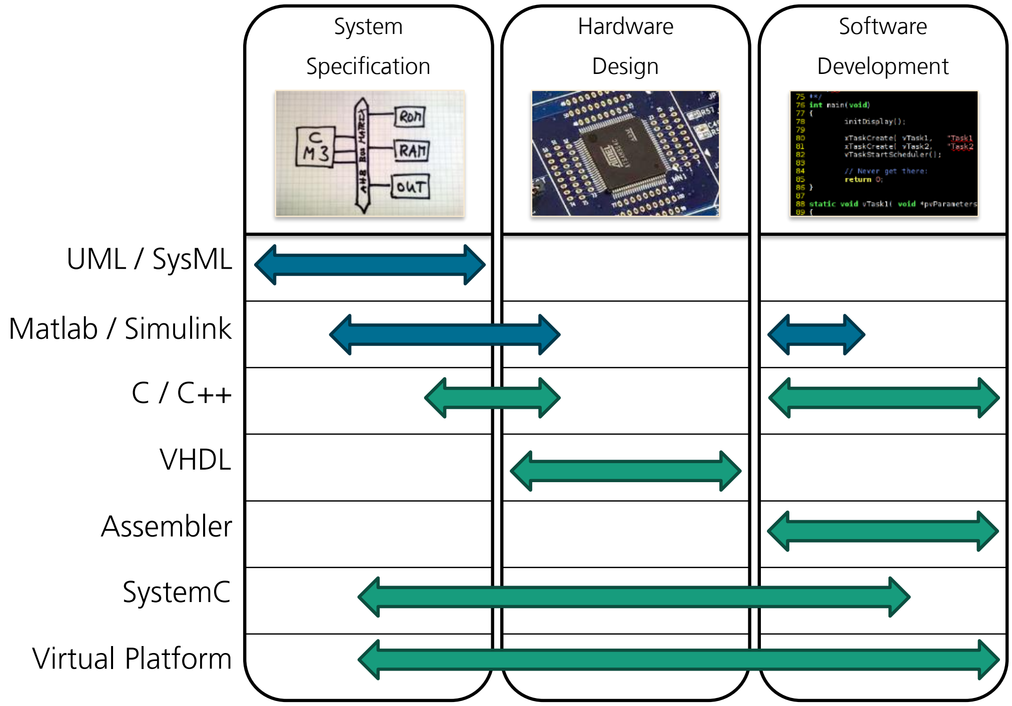
Keys to Make Simulations Fast

- 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



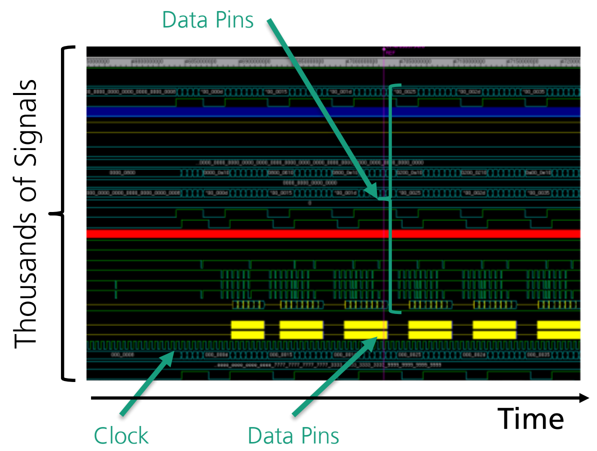
49

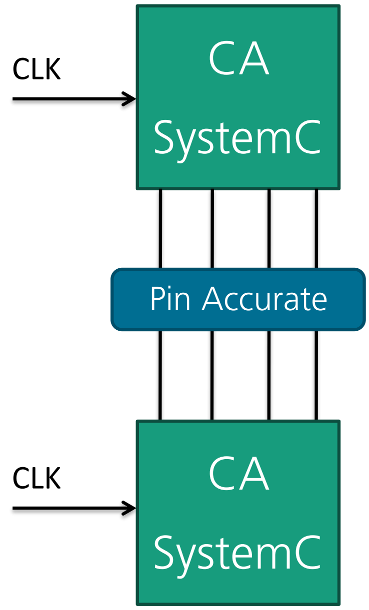
Fraunhofer IESE

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



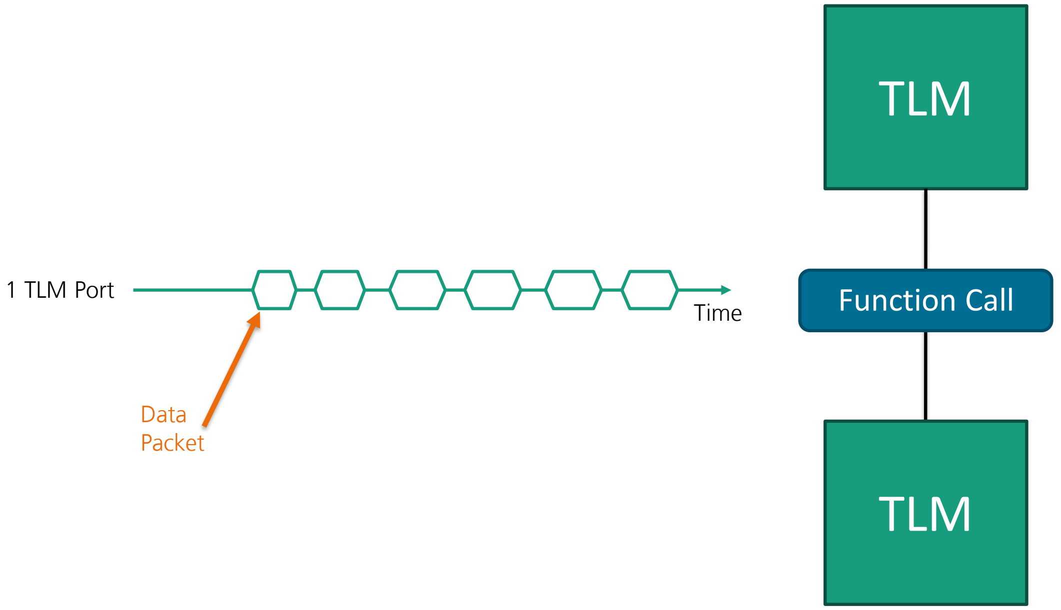


Simulate every event!!

Source: Doulos Ldt. www.doulos.com

Your notes:

Lookout: Transaction Level Modeling (TLM)

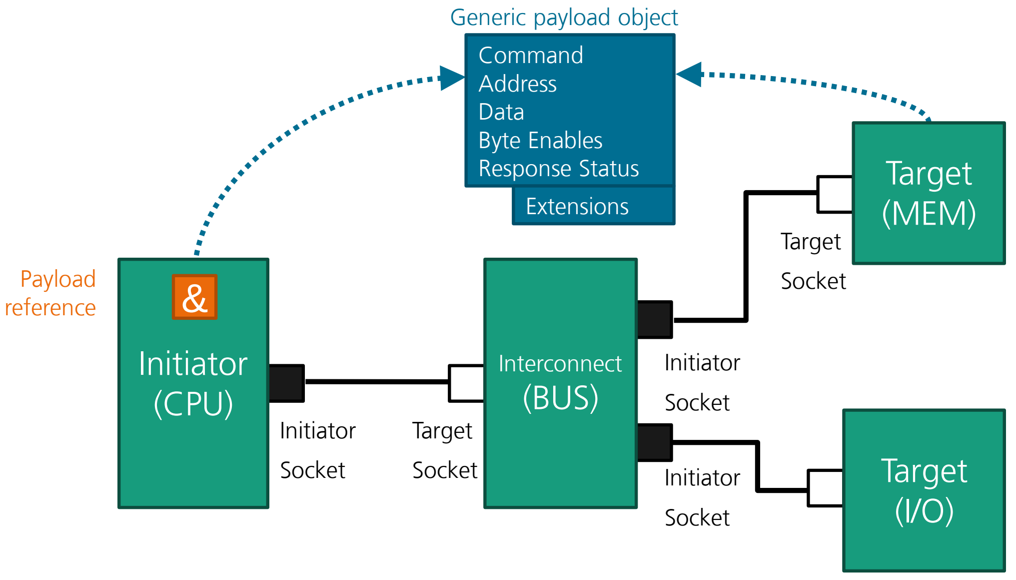


100-10,000 X faster simulation!

Fraunhofer IESE

Your notes:

Lookout: Transaction Level Modeling (TLM)



Fraunhofer IESE

Your notes: