SystemC and Virtual Prototyping

Dr. Matthias Jung, Fraunhofer Institute IESE

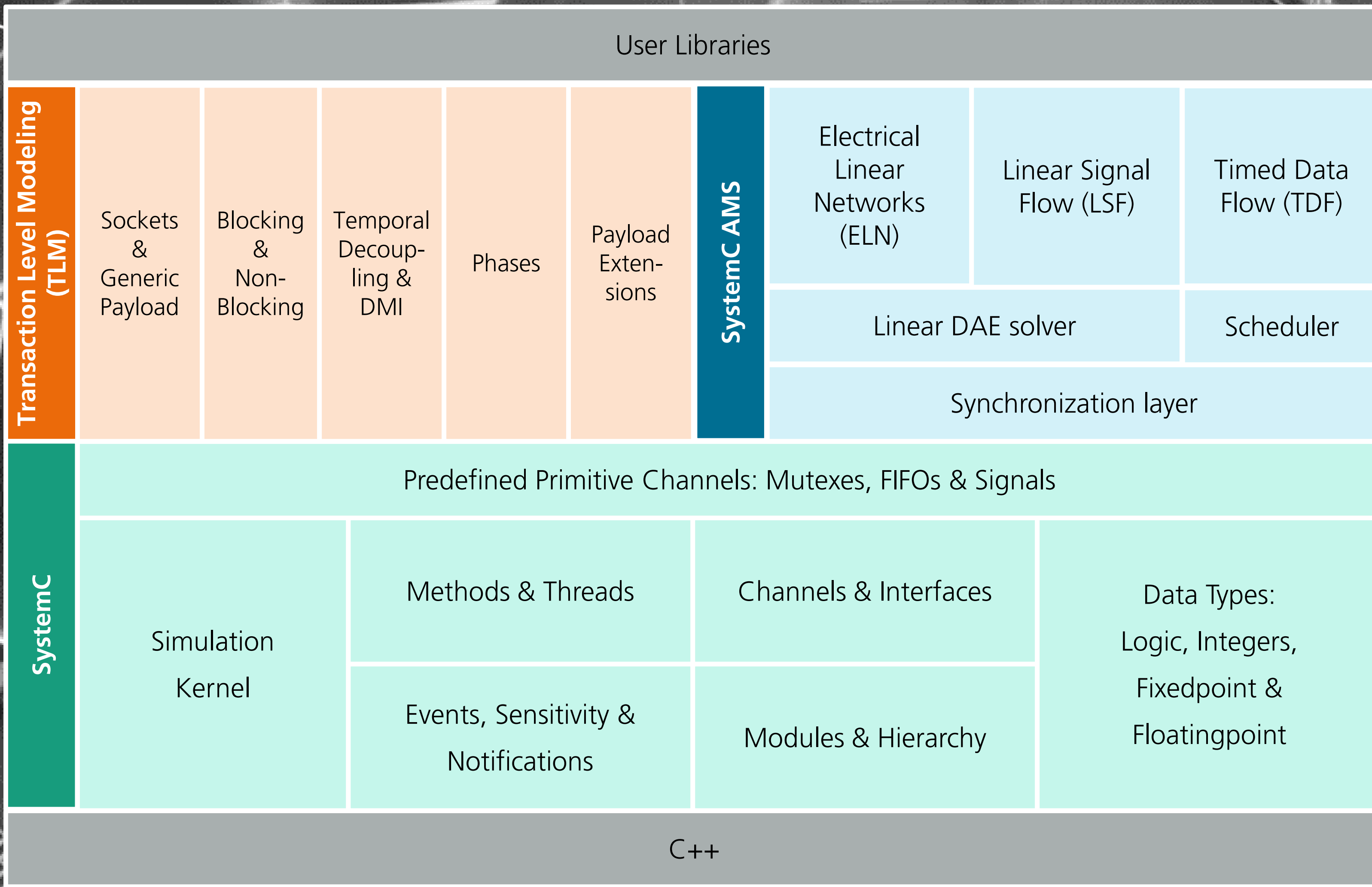
matthias.jung@iese.fraunhofer.de

Fraunhofer IESE

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

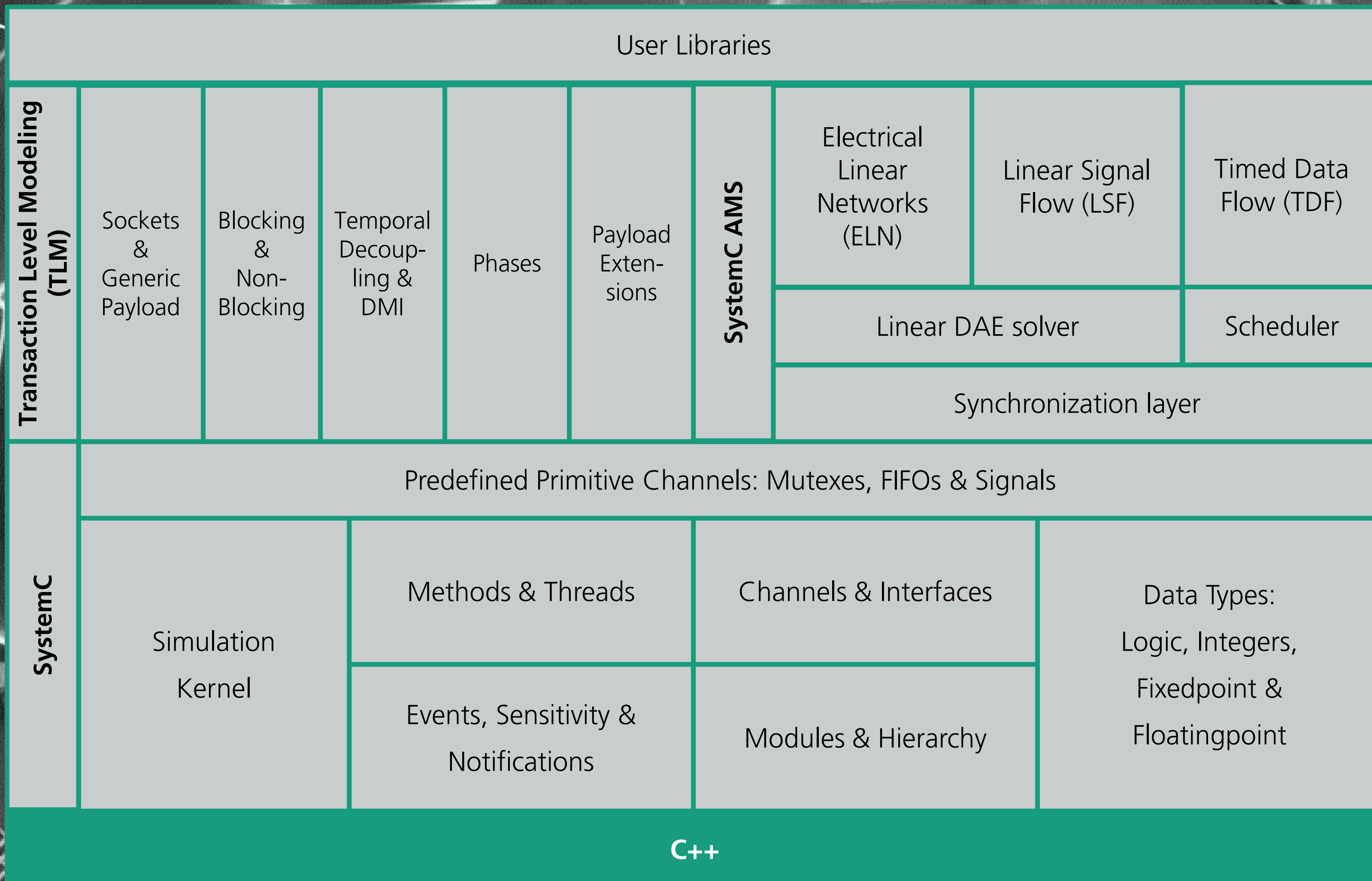

User Libraries
Transaction Level Modeling (TLM)
Sockets & Generic Payload
Blocking & Non-Blocking
Temporal Decoup-ling & DMI
Phases
Payload Exten-sions
SystemC AMS
Electrical Linear Networks (ELN)
Linear Signal Flow (LSF)
Timed Data Flow (TDF)
Linear DAE solver
Scheduler
Synchronization layer
SystemC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

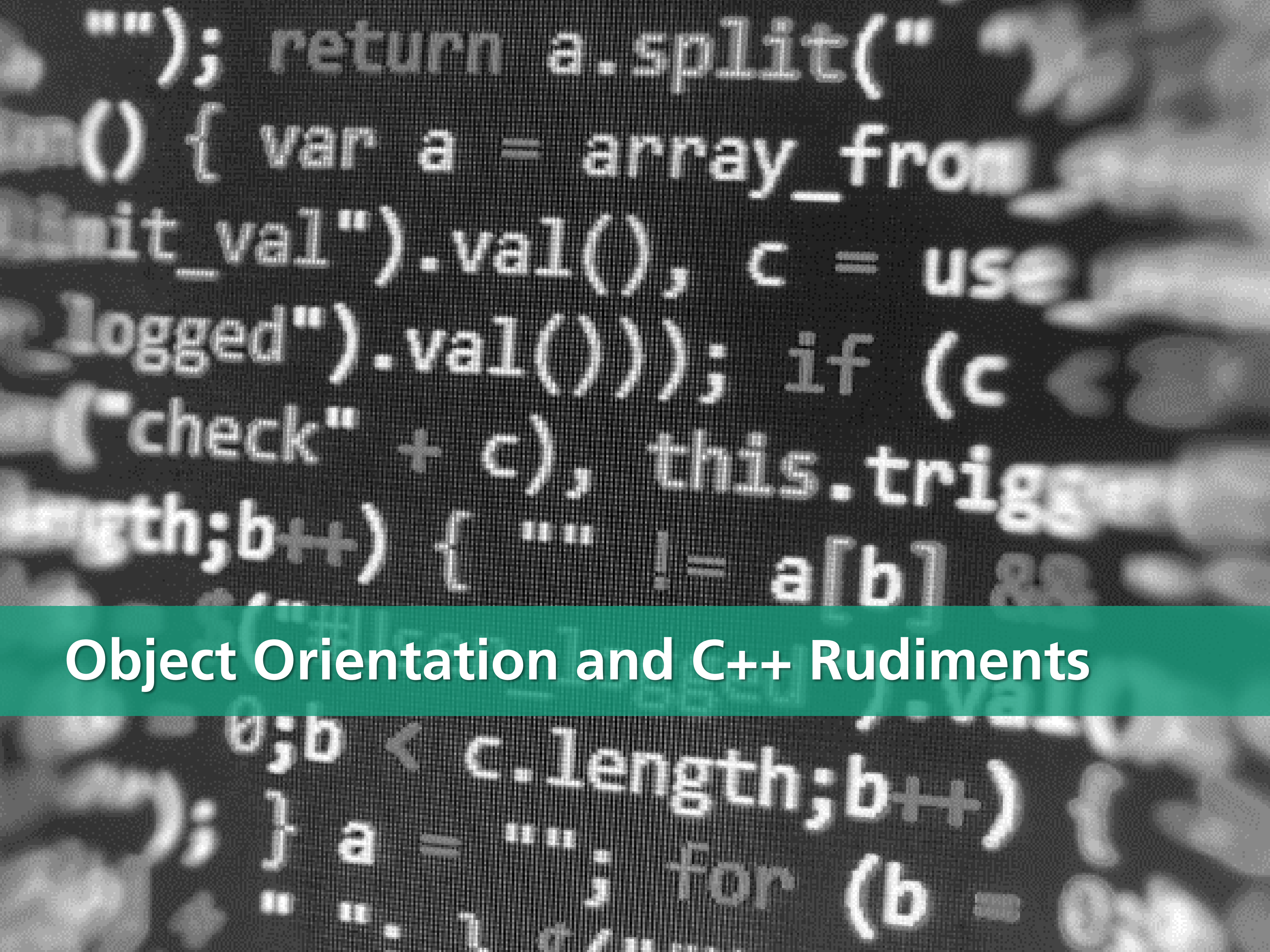
Time Planning

	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

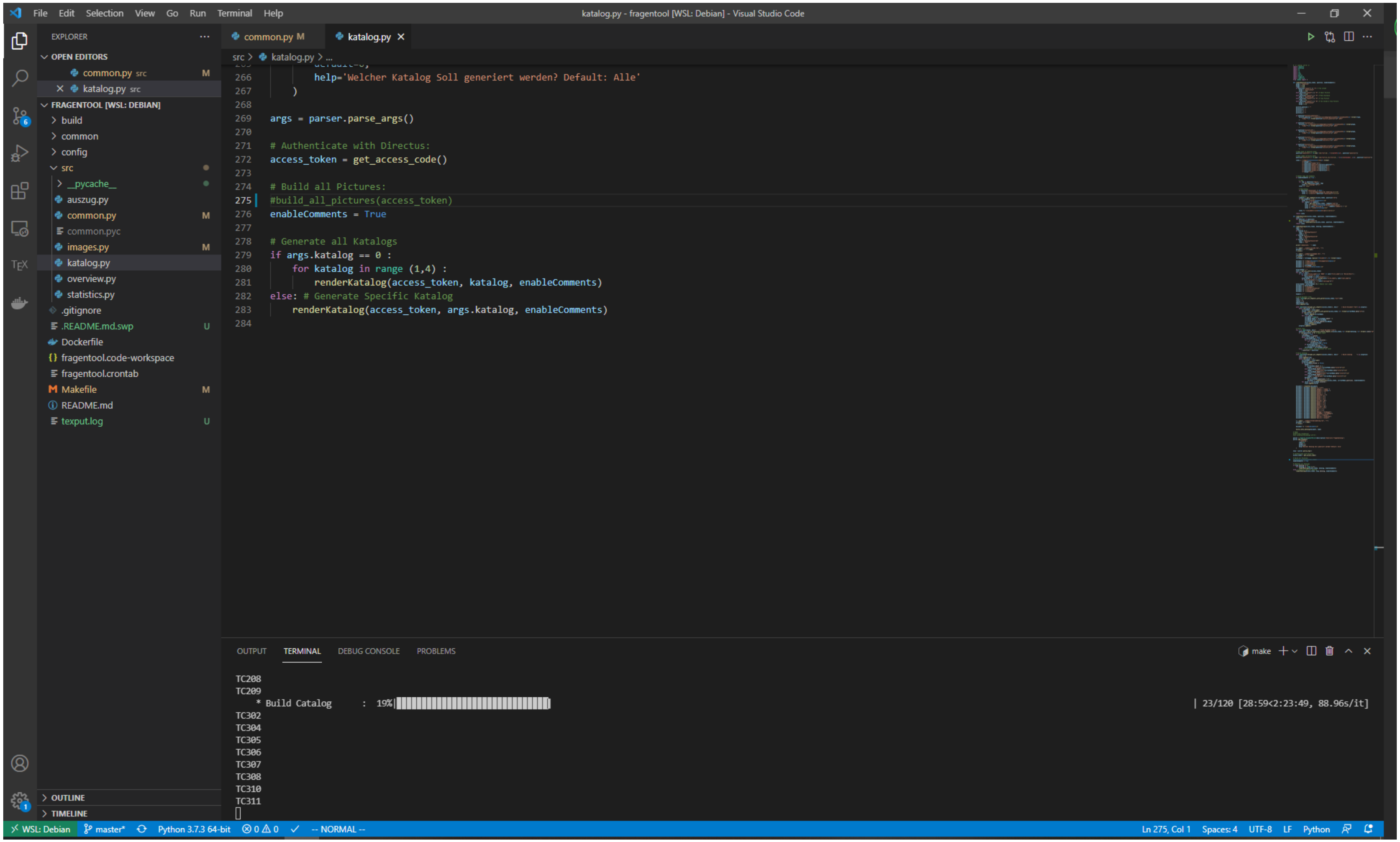
SYSTEMC™


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
Events, Sensitivity & Notifications
Modules & Hierarchy
Data Types: Logic, Integers, Fixedpoint & Floatingpoint
C++



Object Orientation and C++ Rudiments

IDE: VS Code



Fraunhofer IESE

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;
}
```

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);
}
```

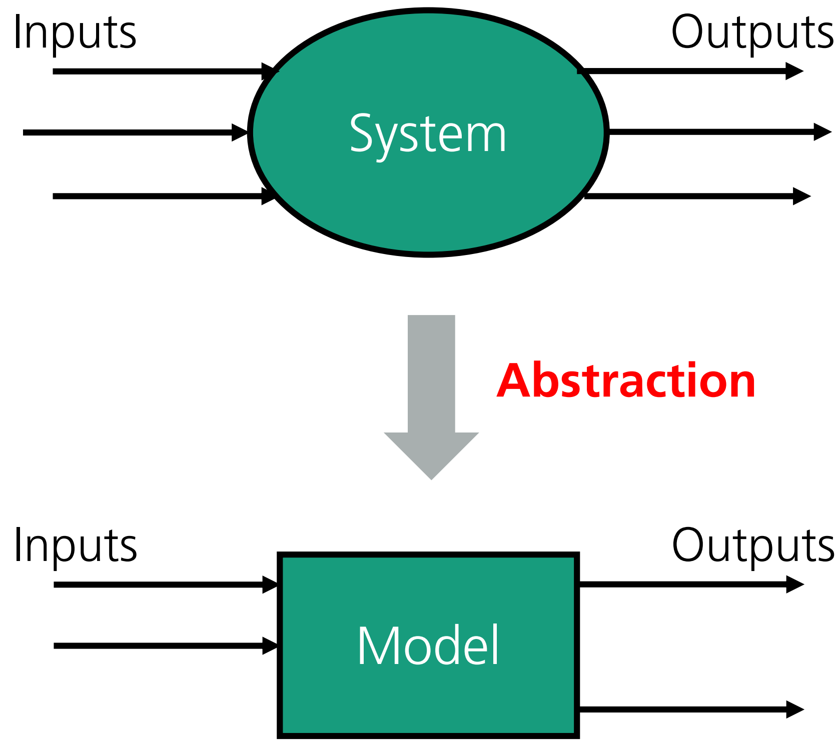
System and Model

- 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

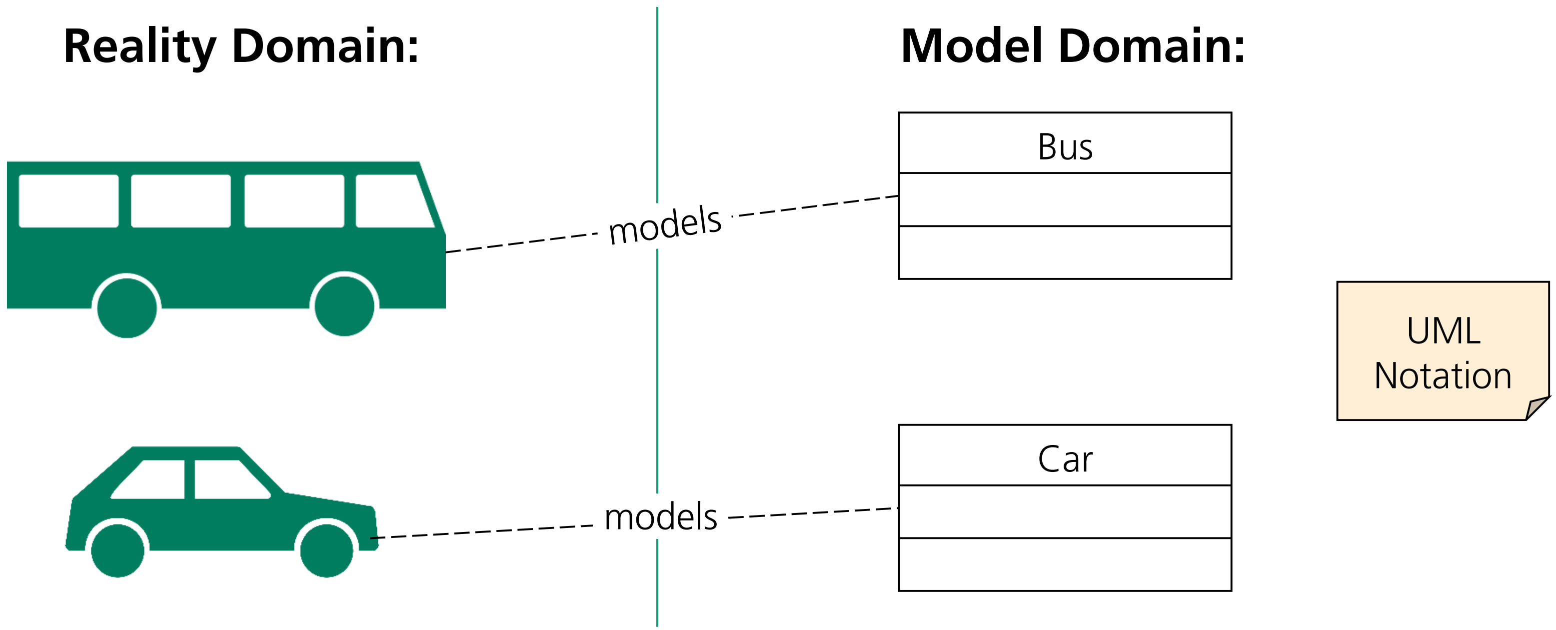


Fraunhofer IESE

Object Orientation (OO)

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

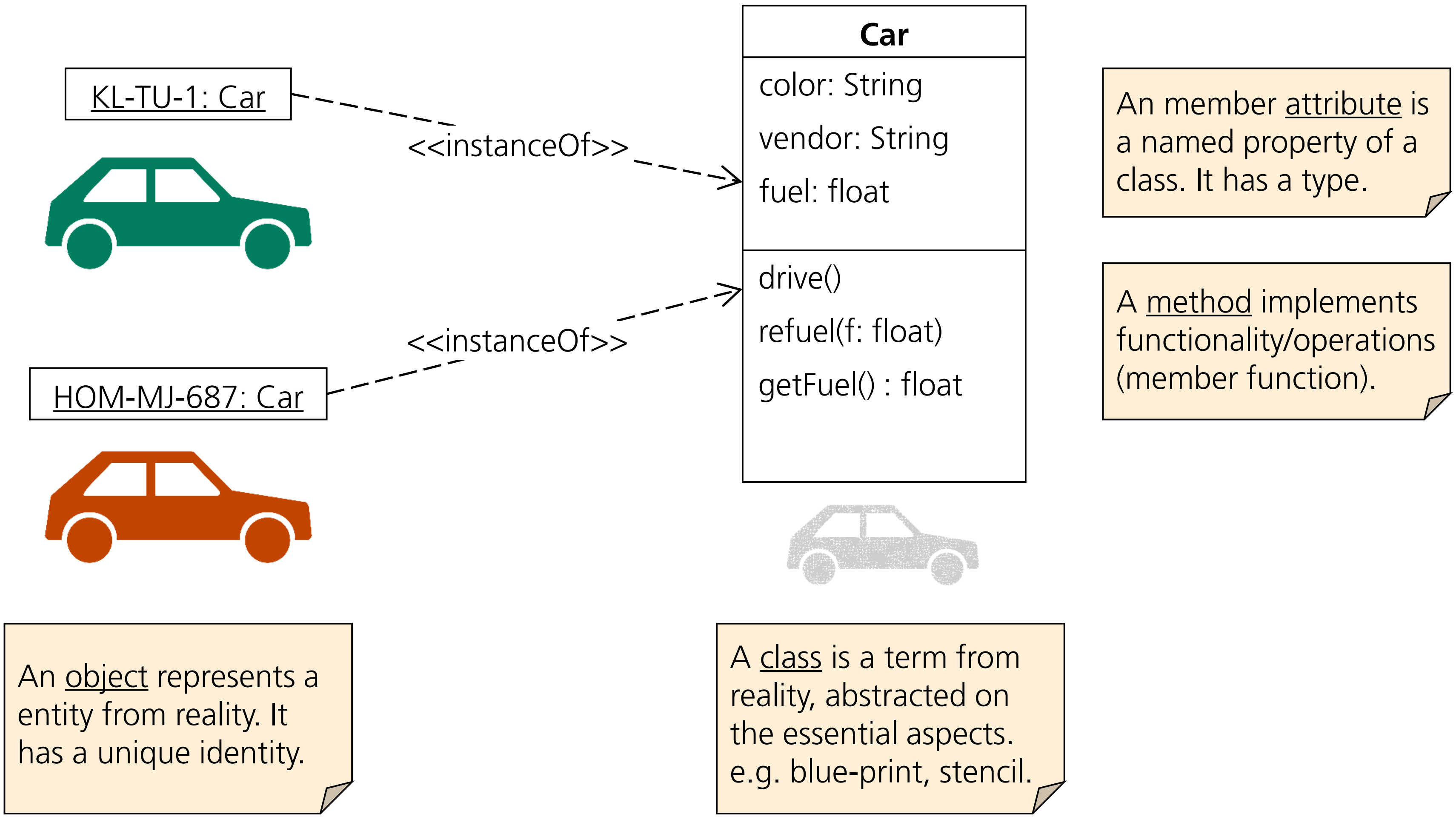
Object Orientation is the alignment on the things of reality!



Object Orientation (OO)

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

Object, Classes, Attributes, Methods



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;
}
```

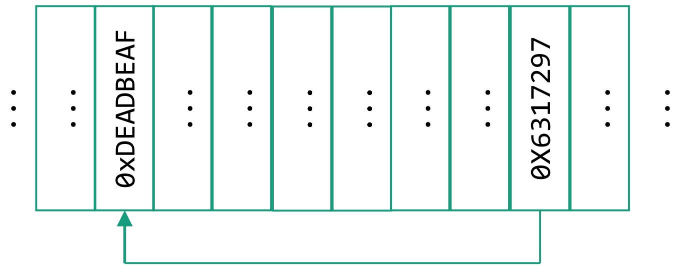
Methods can be defined within the class definition or outside

Pointers (->)

Object of class car are created by passing constructor arguments!

Fraunhofer IESE

Pointers, `new` and `delete`




```cpp
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

Call by Reference and Call by Value

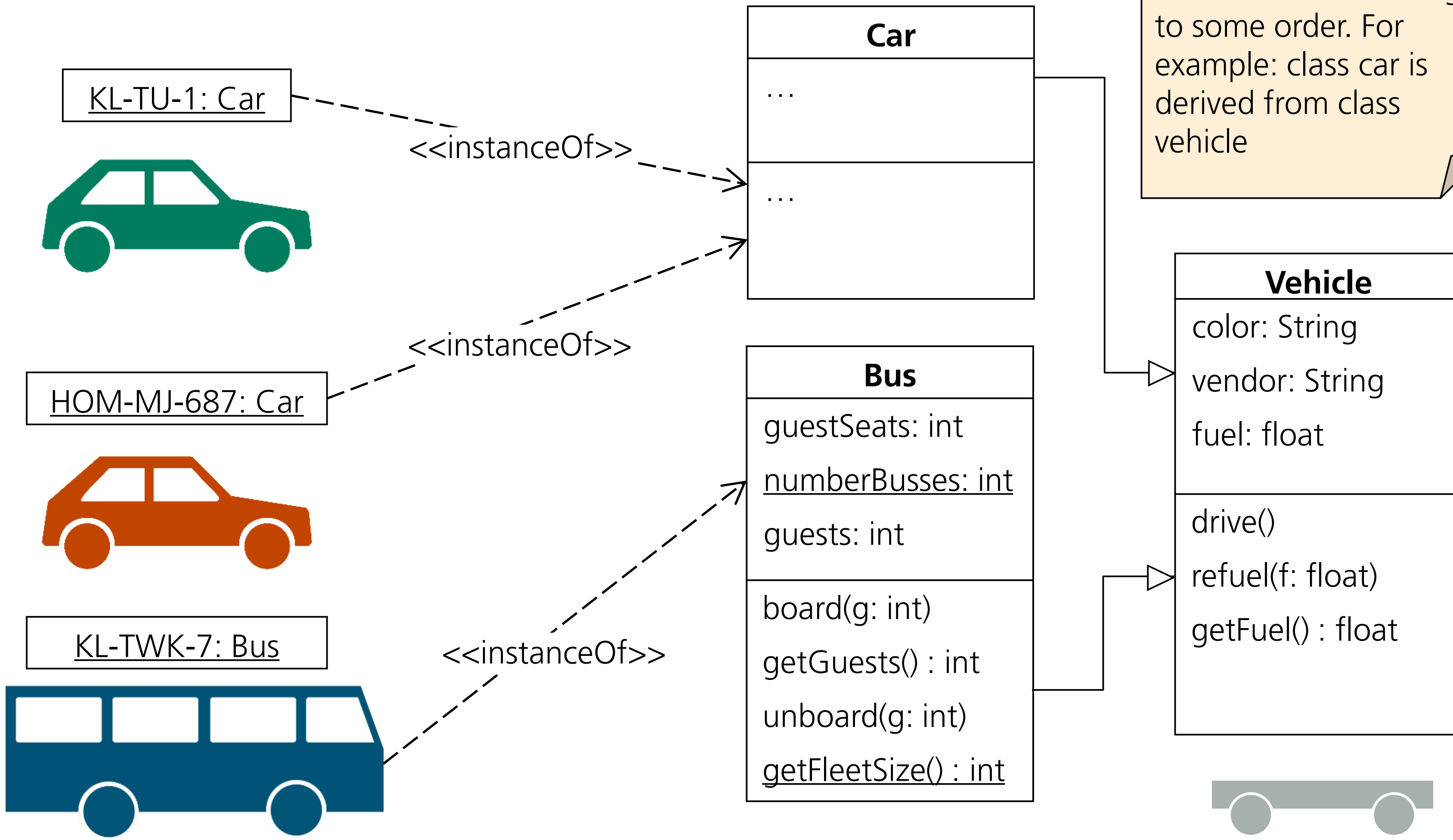
```
#include<iostream>

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

Inheritance and Static Members

Inheritance is the organization of abstractions according to some order. For example: class car is derived from class vehicle



A static member (underlined) is shared by all objects of the class

Fraunhofer IESE

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;
    }
};
```

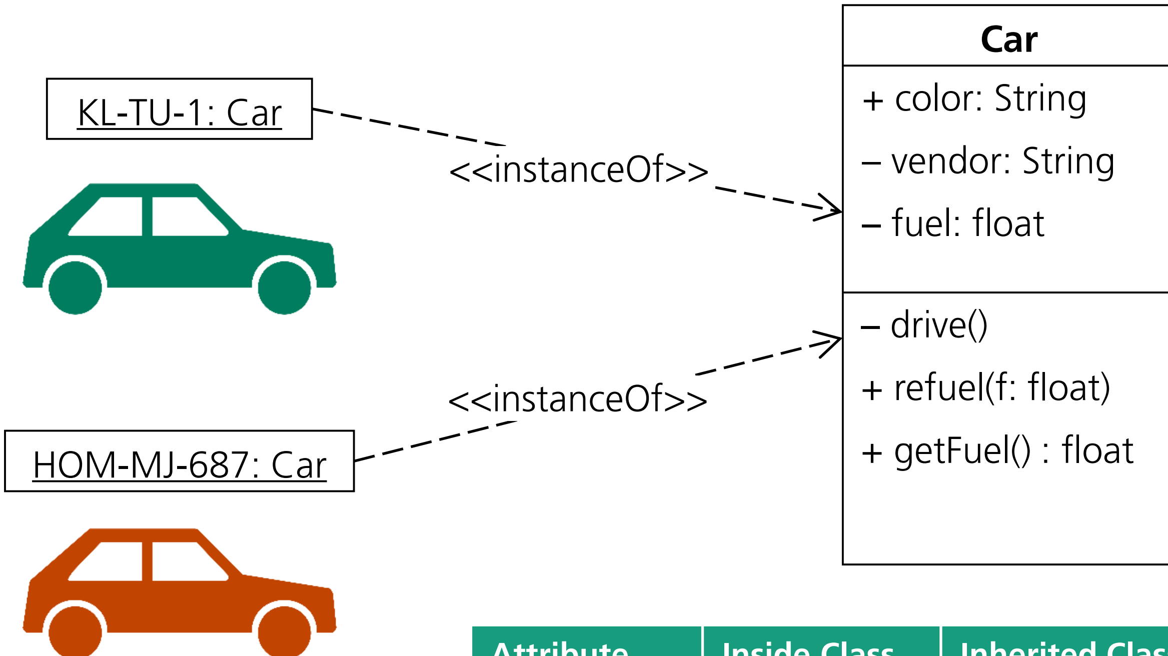
```cpp
class bus : public vehicle {
    // Additional Member Variables:
    public:
    int guestSeats;
    static int numberBusses;
    int guests;

    // 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

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.

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		

Fraunhofer IESE

Access Variables of Parent Class

```cpp
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
}
```

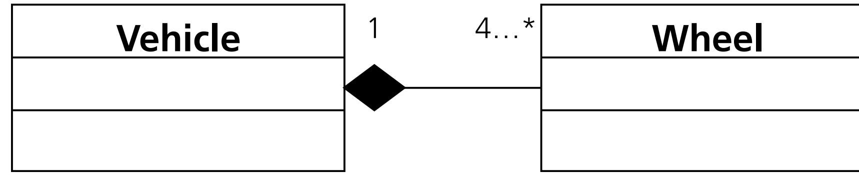
- The *private* member are not accessible outside the class.
- They can be accessed only trough methods of the class.
- The *public* members are accessible outside the class.
- The *protected* members can be accessed by inherited classes.
- Parental member can be accessed with the parents class name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

Fraunhofer IESE

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

```cpp
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;
      }
};
```

```cpp
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

Polymorphism – Motivation

```cpp
#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;
      };
};
```

```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;
}
```

& 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!

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

Fraunhofer IESE

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 <u>must</u> implement virtual function! Otherwise compiler crashes!
- Why using it? For structuring! For defining <u>Interfaces → Exchangeability during Runtime</u>

Fraunhofer IESE

Operator Overloading

```cpp
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;
      }
};
```

```cpp
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.

Templates

```cpp
#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

Templates

```cpp
#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

```cpp
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.

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.

Fraunhofer
IESE

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

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/

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());
```

C++ 11, 14 ...

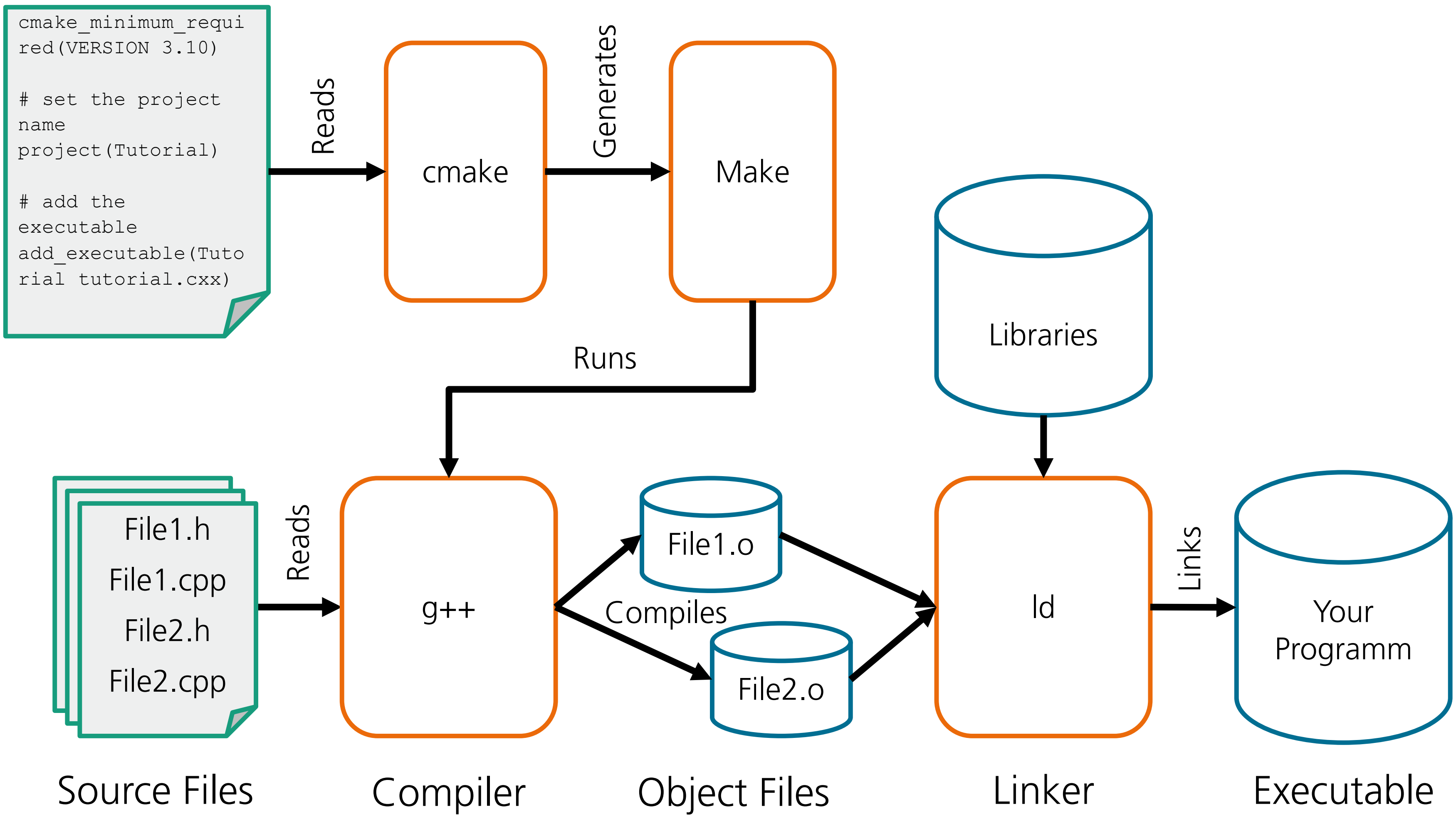
```cpp
#include <iostream>
#include <vector>

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

Configure your project with CMake



Fraunhofer IESE

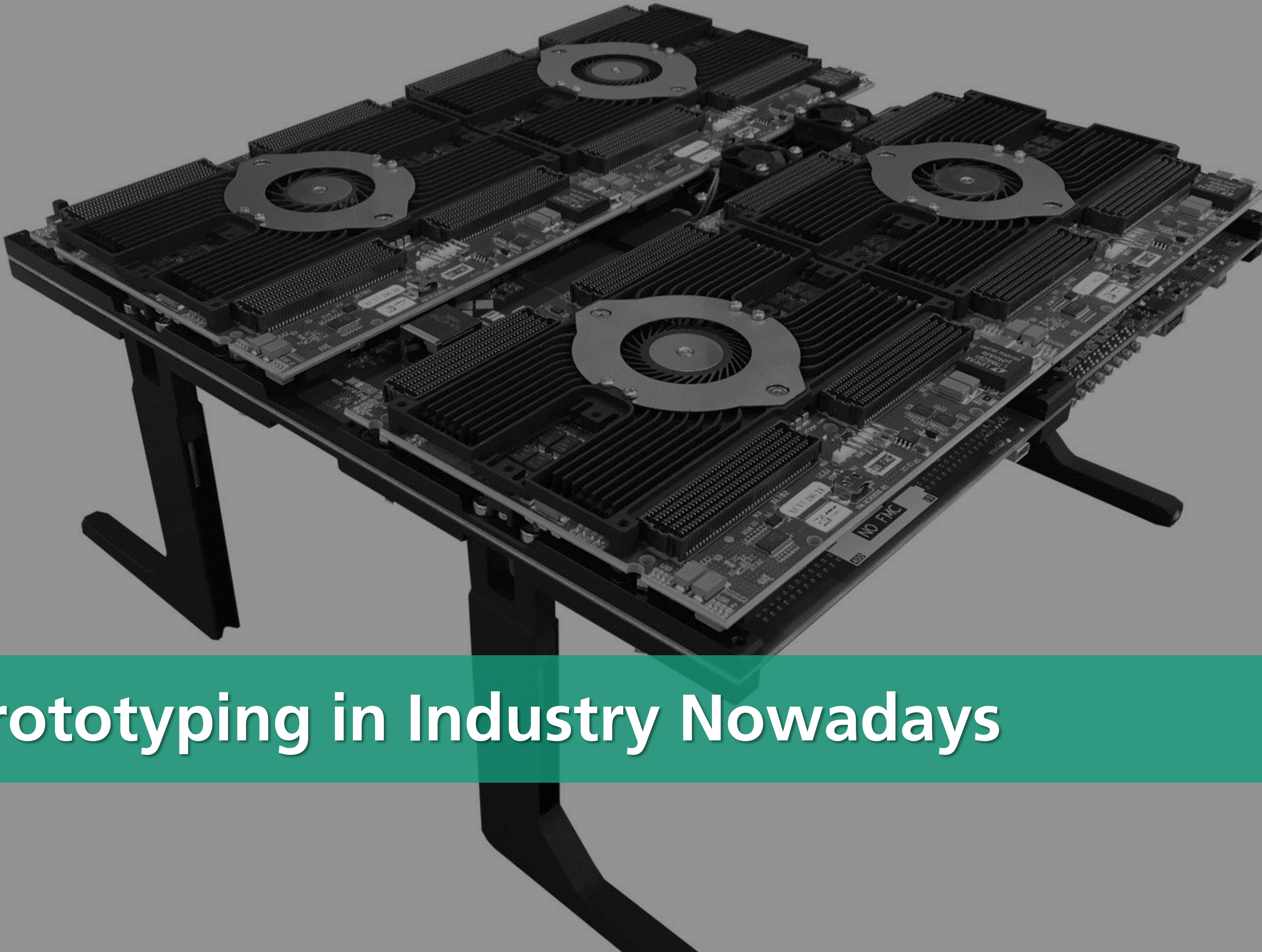
SystemC and Virtual Prototyping

Dr. Matthias Jung, Fraunhofer Institute IESE

matthias.jung@iese.fraunhofer.de

Fraunhofer
IESE

Prototyping in Industry Nowadays



State of the Art – Prototyping Example

1st demonstration to customer

2nd demonstration to customer

Chip Development

Software Development

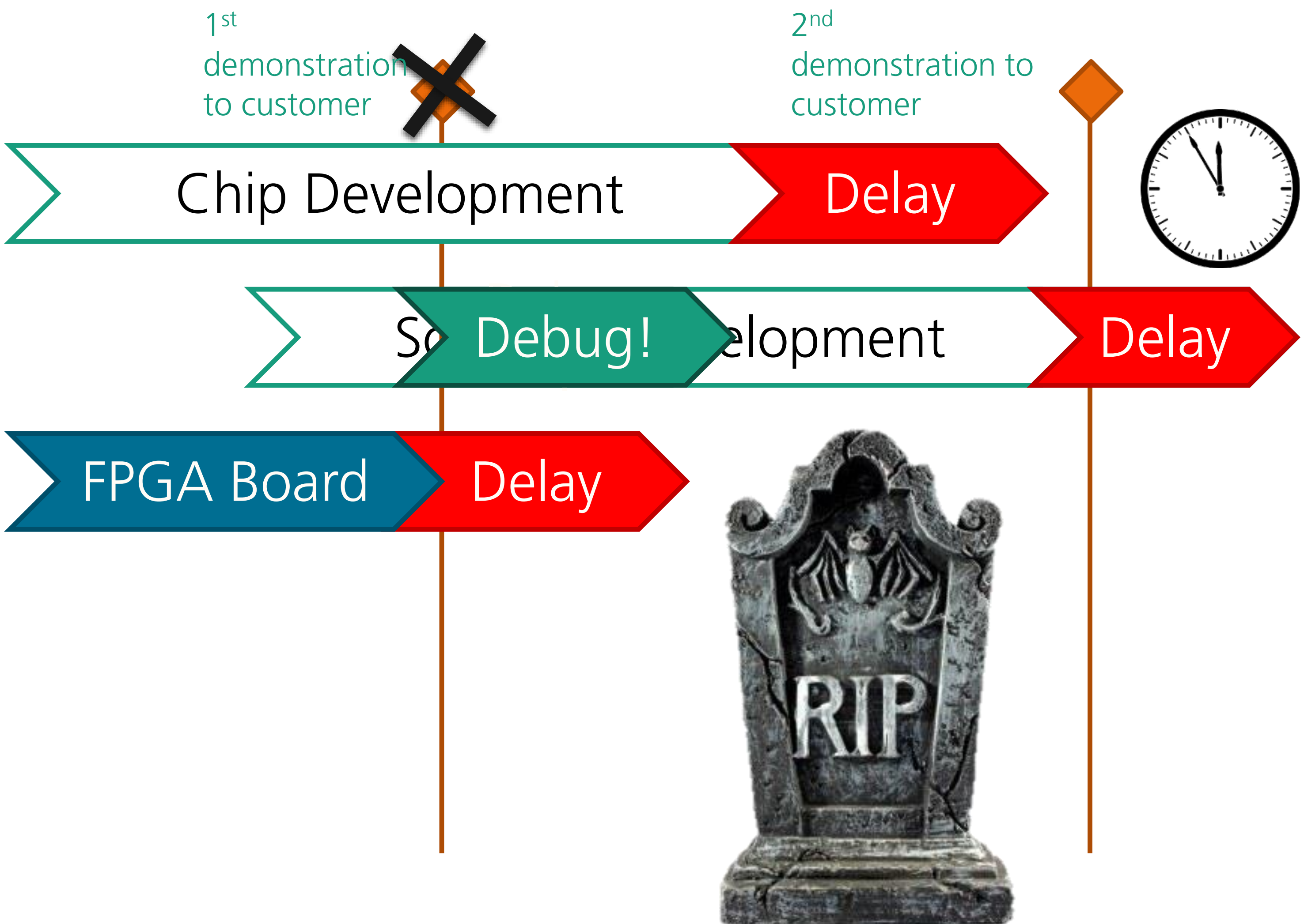
FPGA Board

FUJITSU

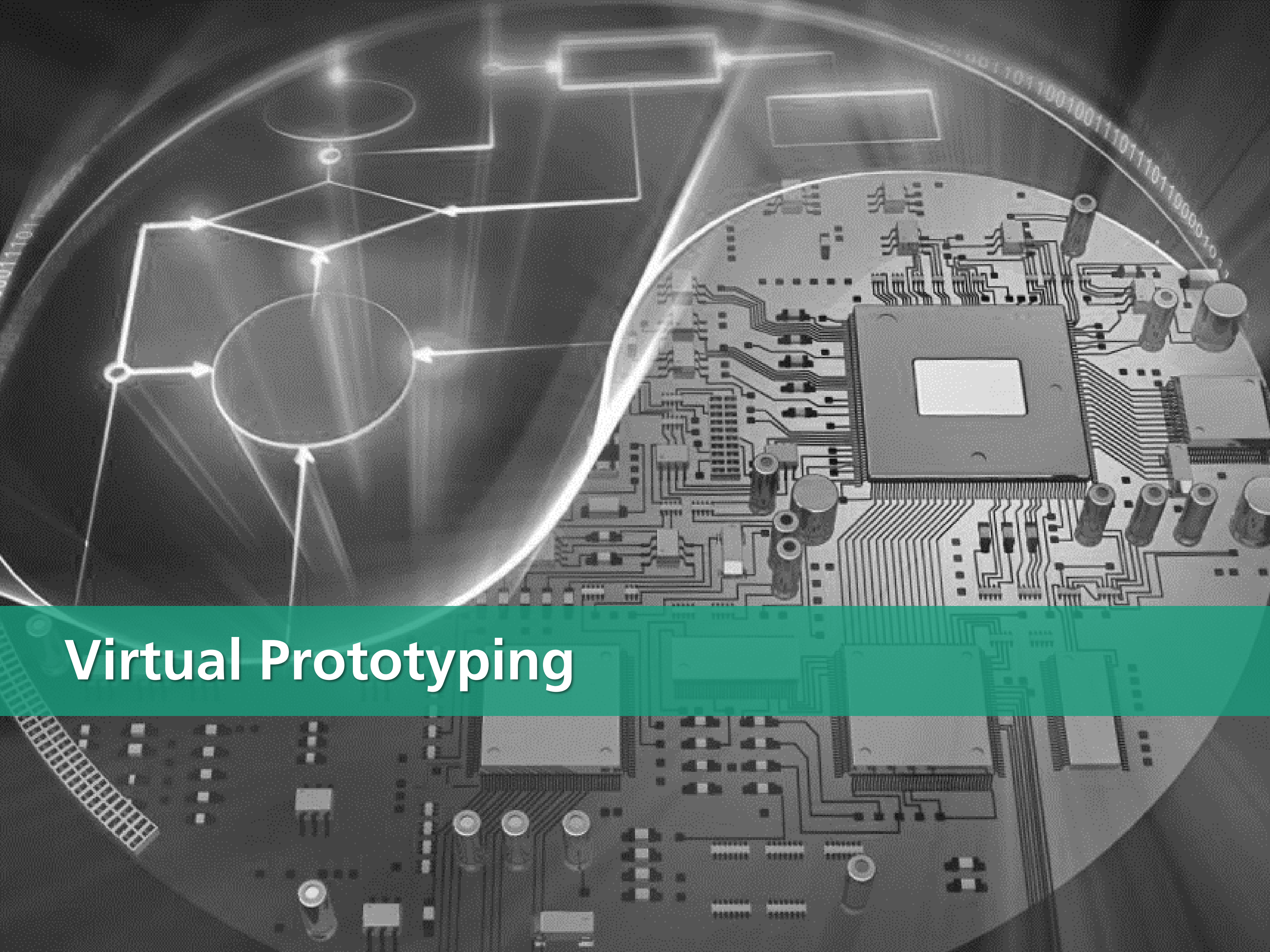
Automotive Graphics Controller SoC MB86R1x 'Emerald'

Source: FUJITSU 2013

State of the Art – Prototyping Example - Reality

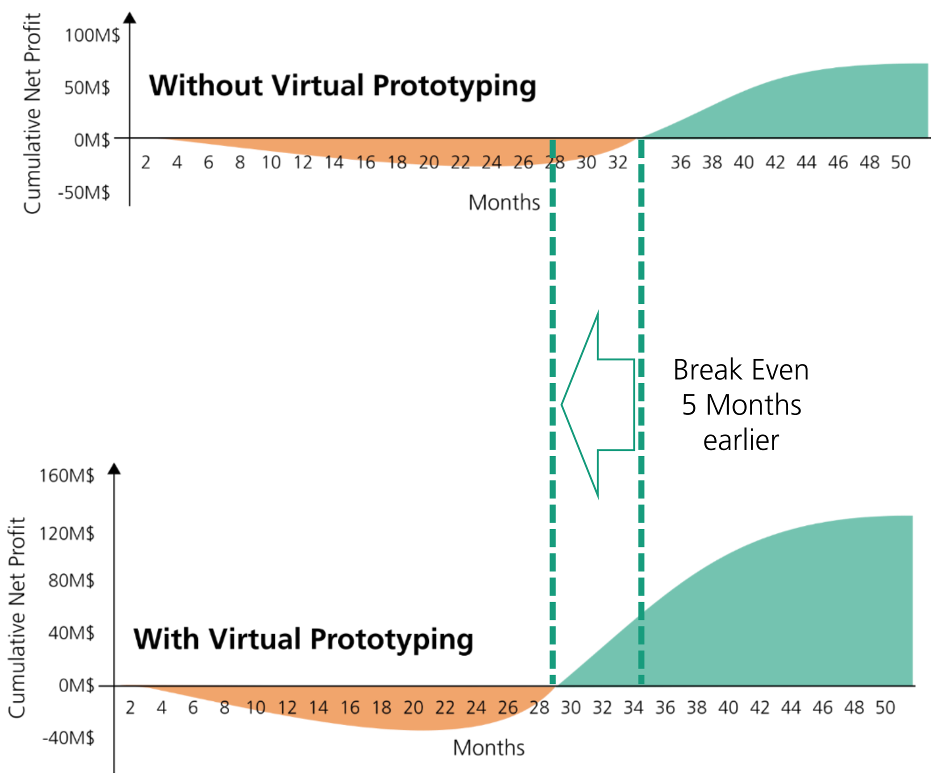


Source: FUJITSU 2013



Virtual Prototyping

Shift Left with Virtual Prototyping

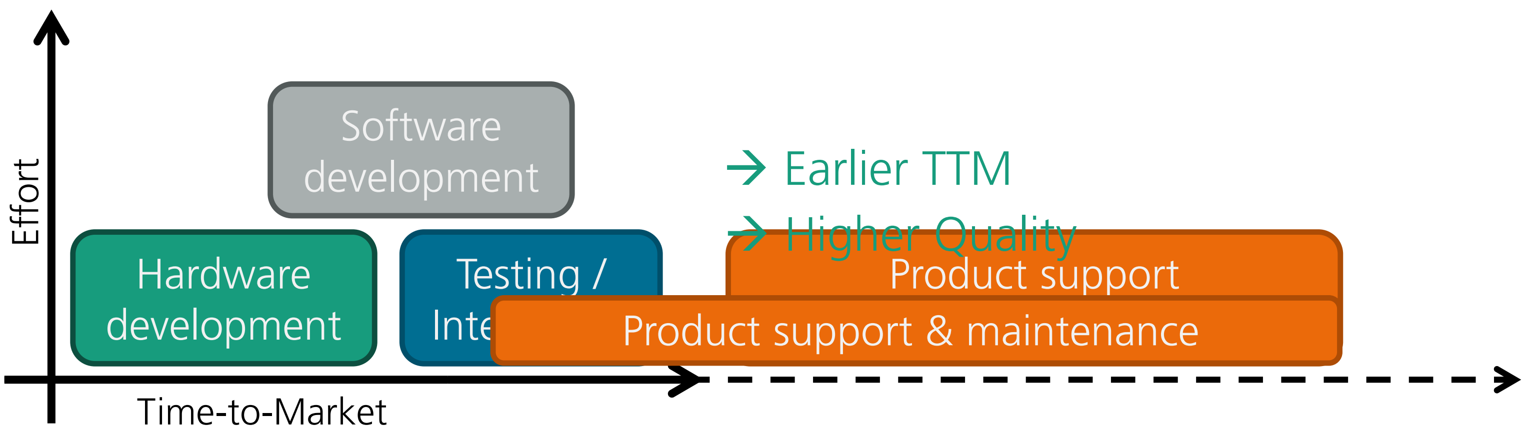


- Hedge decisions early
- Bugs are found and fixed early (Typically 56% of bugs emerge due to wrong requirements)
- Saving Time to Market (TTM) and resources
- Allowing good test coverage
- Better team work with HW engineers, SW developers and testers – this model minimizes frictional differences between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

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

Virtual Prototypes

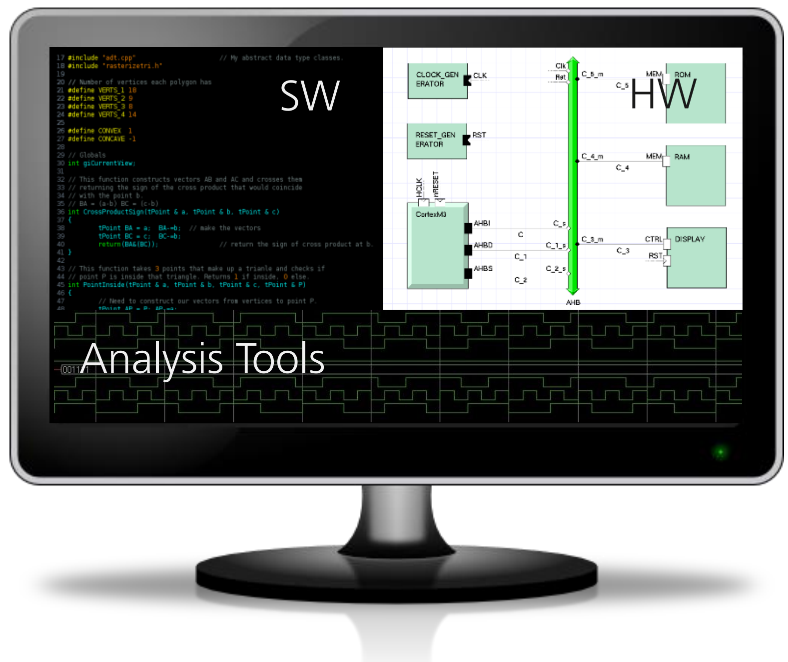
- High-speed functional software models of physical hardware
- Concurrent HW and SW development
- Design Space Exploration (DSE) for HW
- Visibility and controllability over the entire system
- Powerful debugging and analysis tools
- Reuse of components for future projects
- Teams can use it world wide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



Fraunhofer IESE

Move to Virtuality?

Everything is in the Developer's Desktop



Virtual Prototyping will Help!

Chip Development

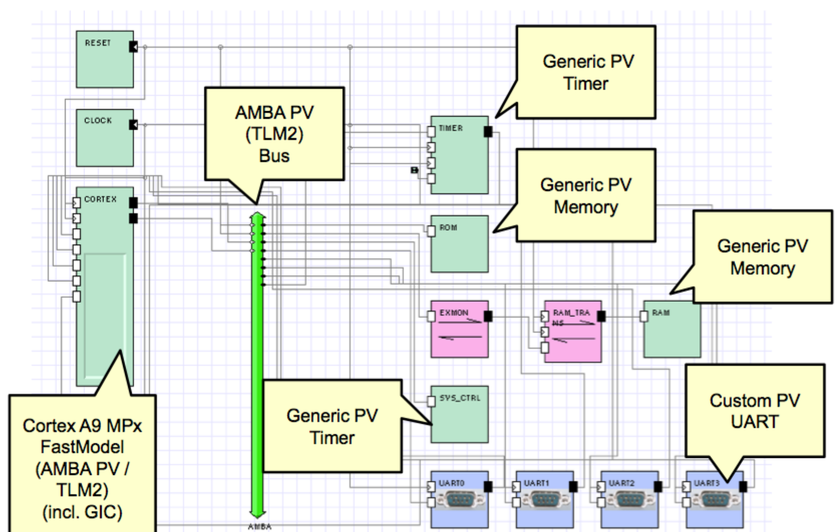
Software Development

Virtual Prototype

2 days after silicon 'Quake' was running

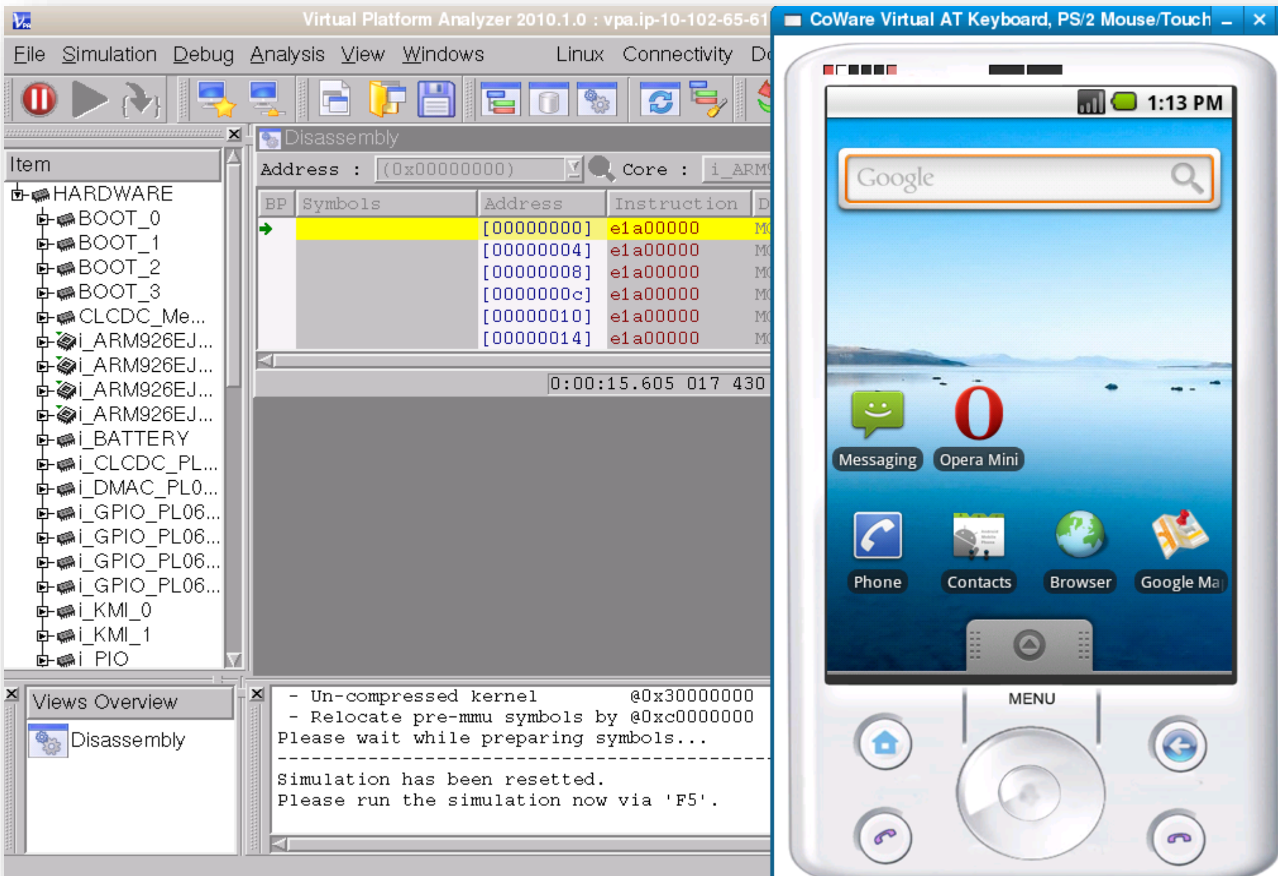
FUJITSU

Automotive Graphics Controller SoC MB86R1x 'Emerald'



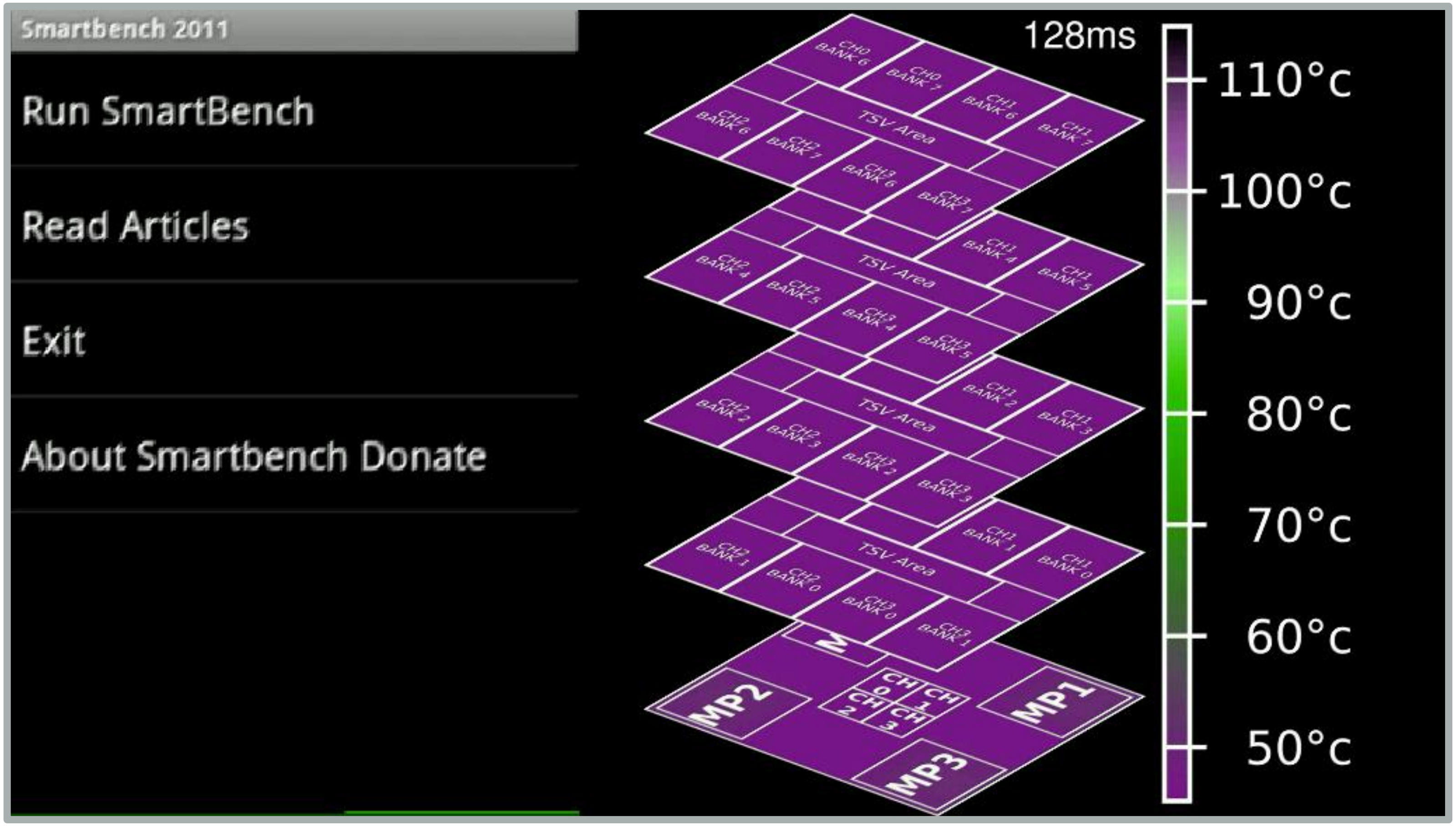
Fraunhofer IESE

How to build a virtual Smartphone?



Fraunhofer IESE

Simulate Future Smartphone with 3D Integrated Circuits



Fraunhofer
IESE

Accuracy vs. Speed Trade-Off

Fraunhofer IESE

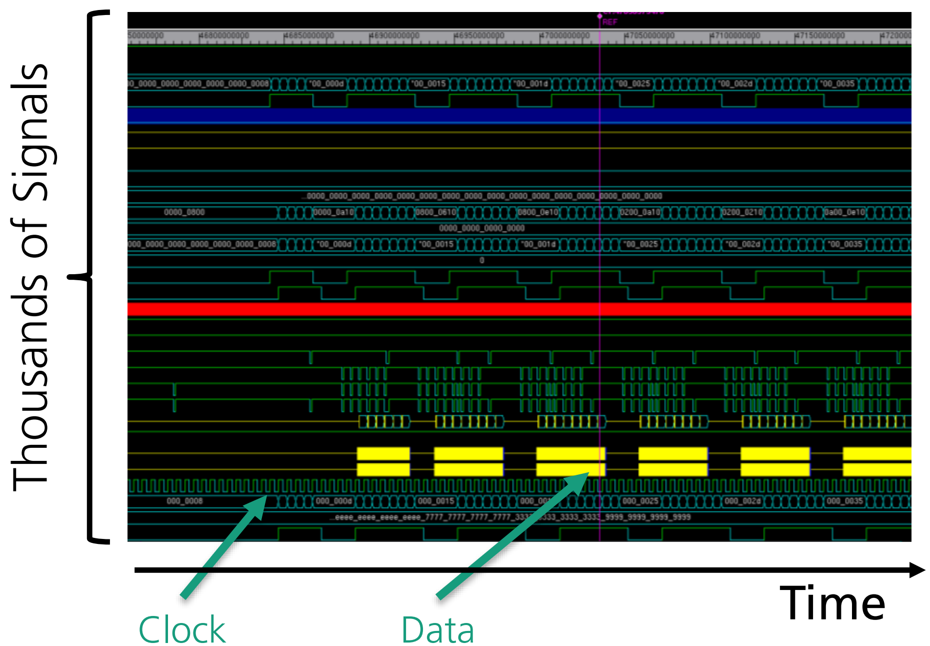
Accuracy vs. Speed Trade-Off

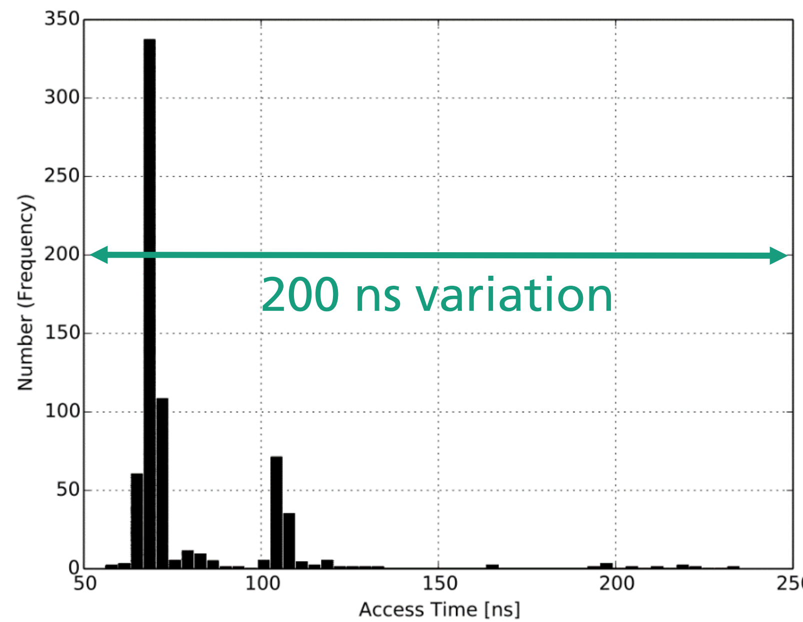
E.g. RTL Simulations:

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

E.g. System Level Simulations:

- Fast
- Large flexibility
- Inaccurate



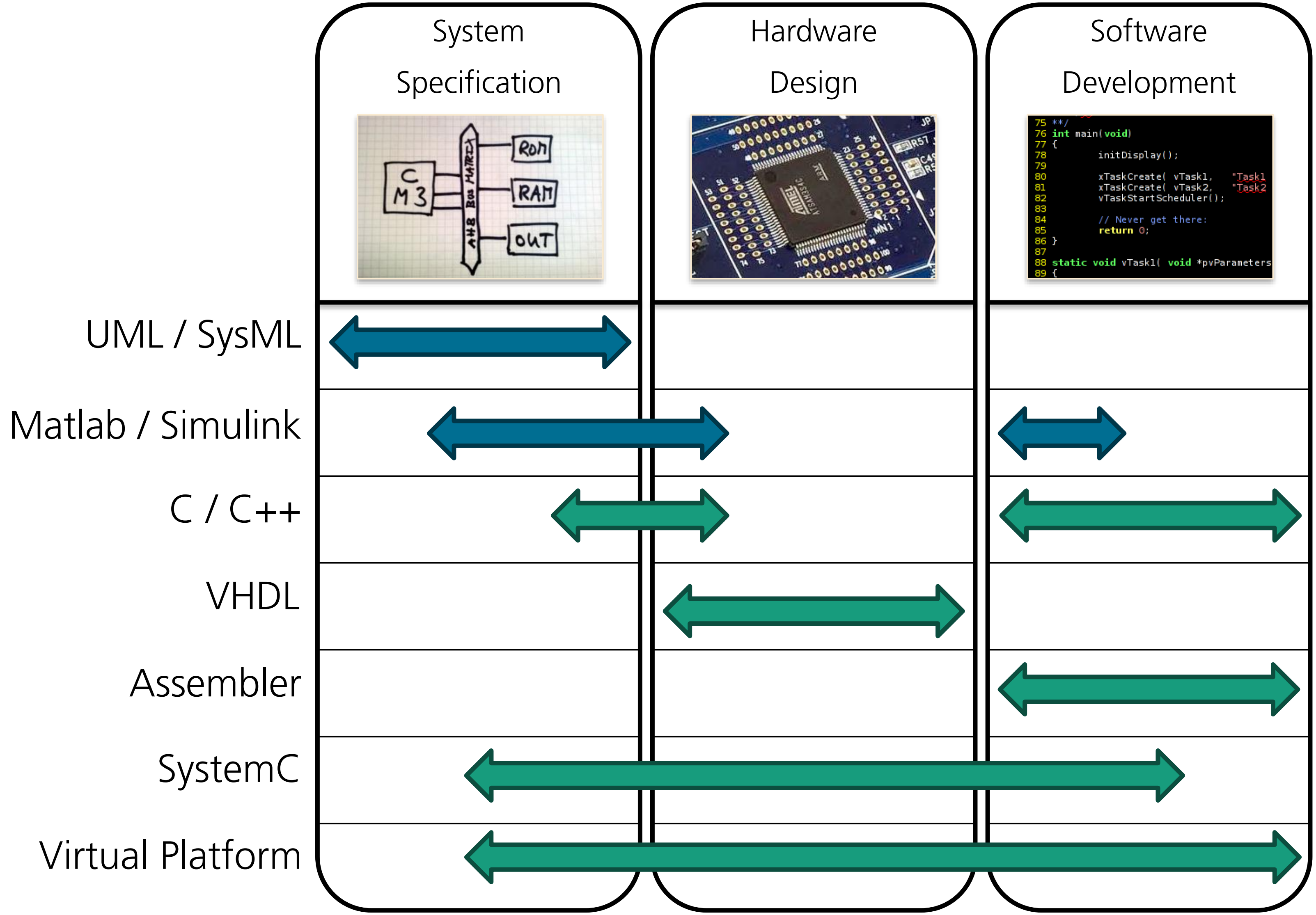


Fraunhofer IESE

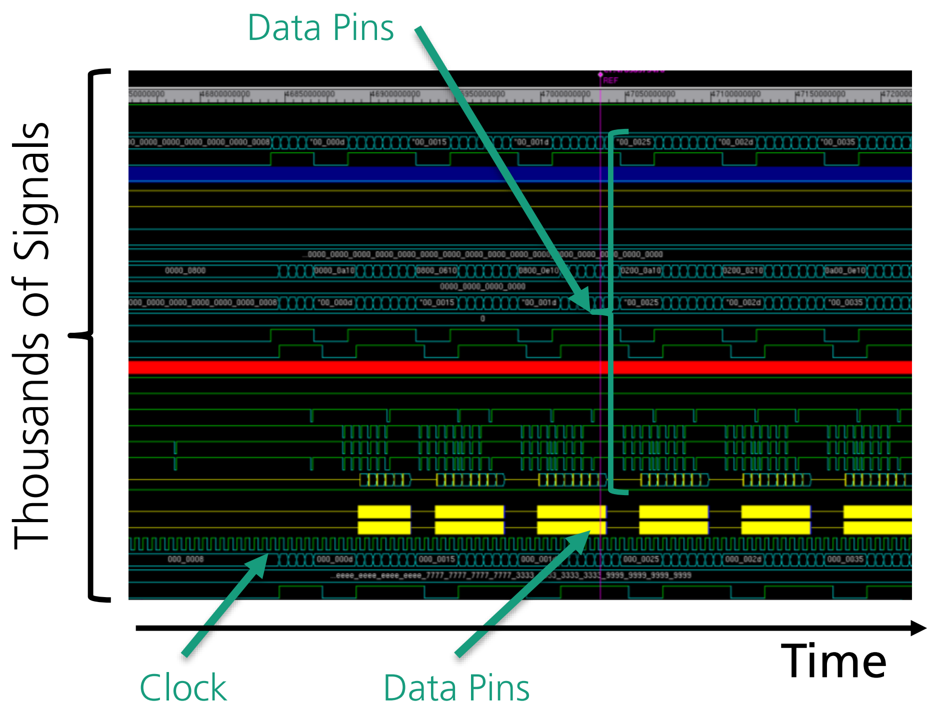
Keys to Make Simulations Fast

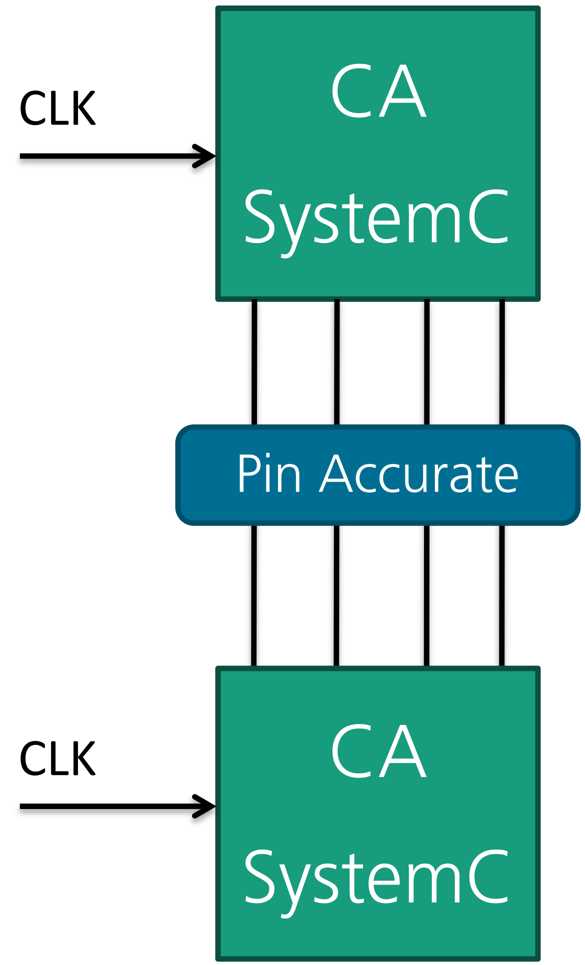
- Be certain about what you want to model
- Ask you, which details are really important?
- Technical Aspects:
  - Saving time simulation, events and clocks (simulate only important events!)
  - Avoid moving large amounts of data
  - Avoid simulation context switches i.e. limit the number of calls to `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,…>`)

Different Modelling Languages



Lookout: Cycle Accurate Simulation



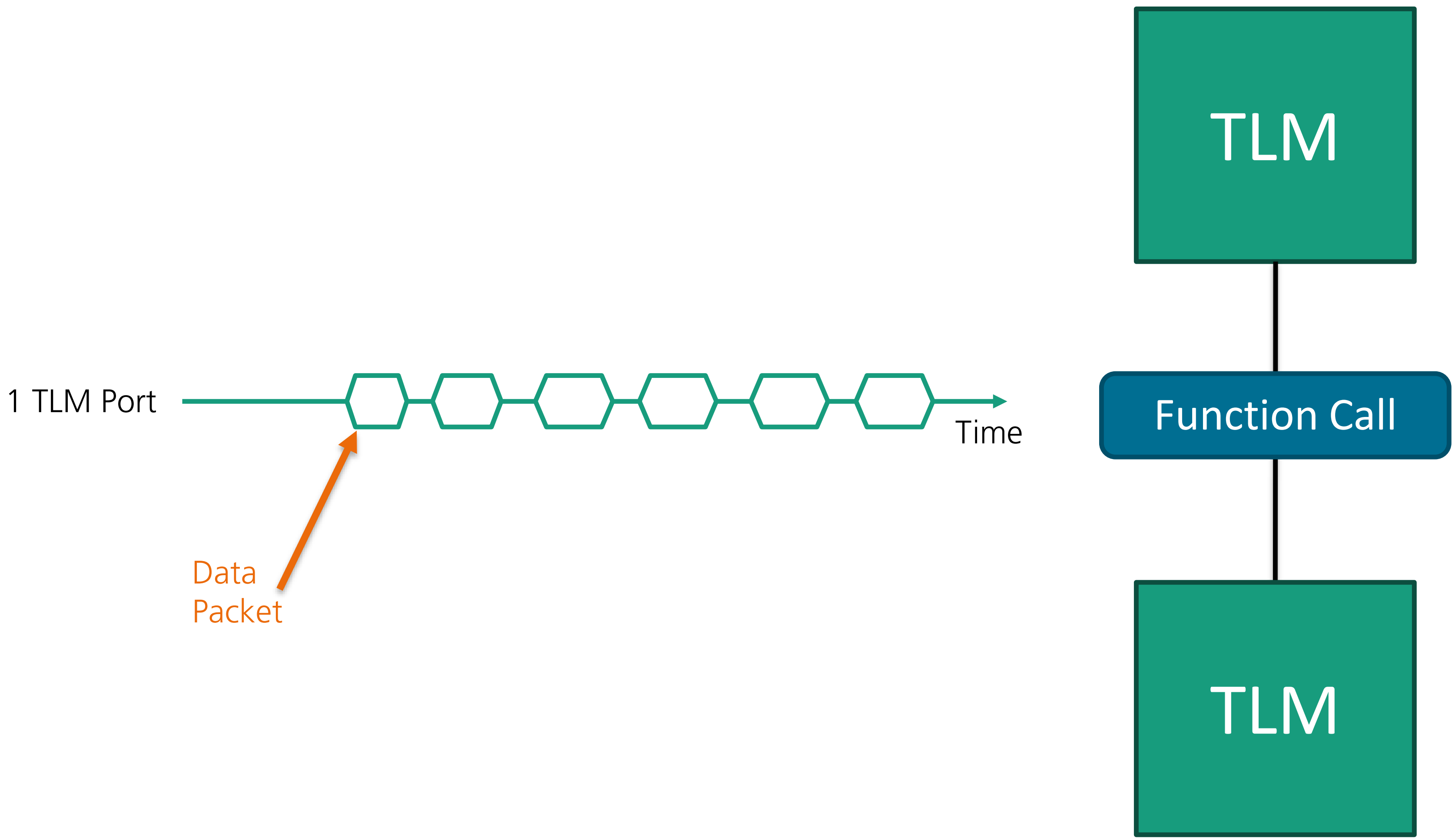


Simulate every event!!

Source: Doulos Ldt. www.doulos.com

Fraunhofer IESE

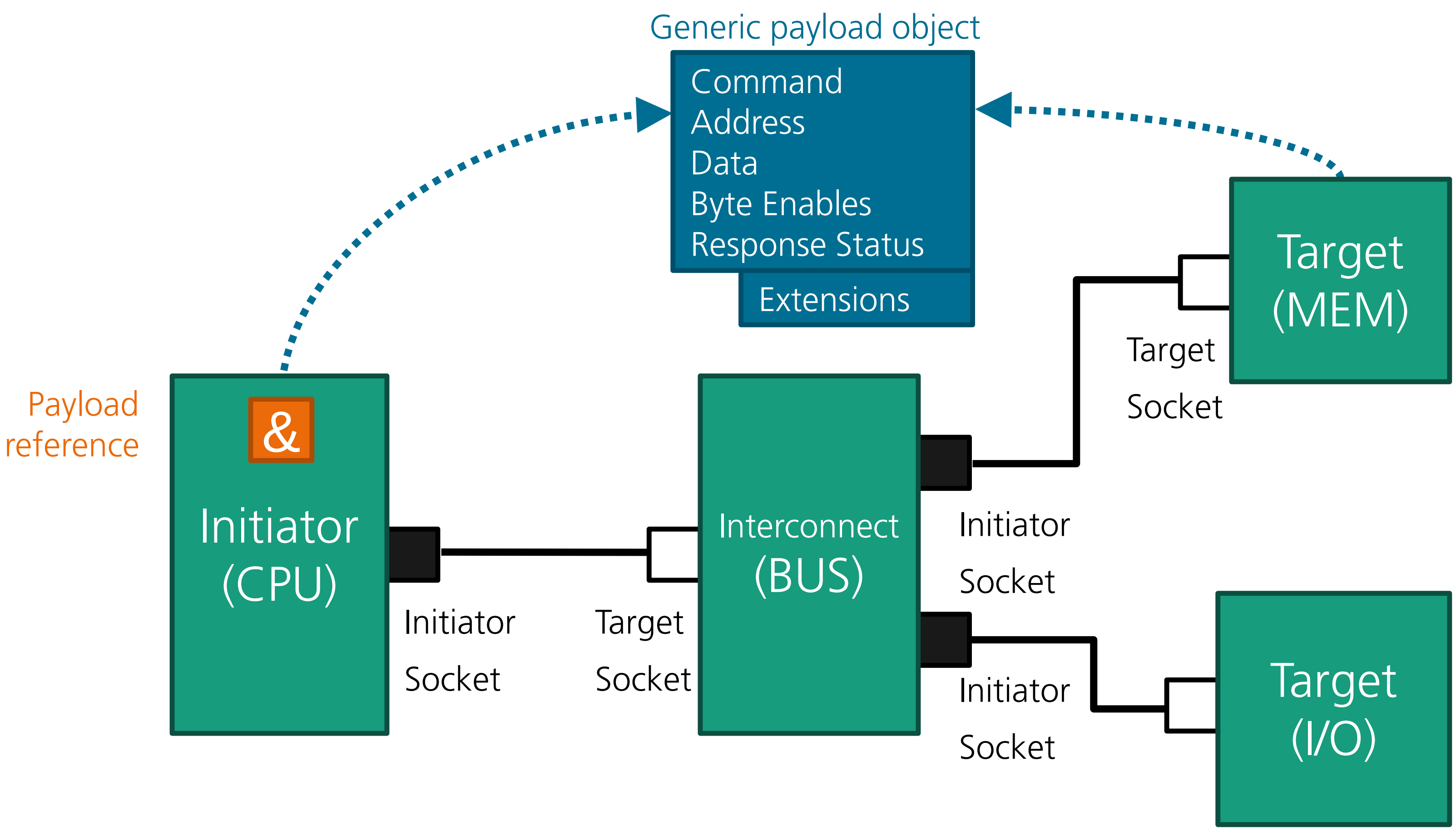
Lookout: Transaction Level Modeling (TLM)



100-10,000 X faster simulation!

Fraunhofer IESE

Lookout: Transaction Level Modeling (TLM)



Fraunhofer IESE