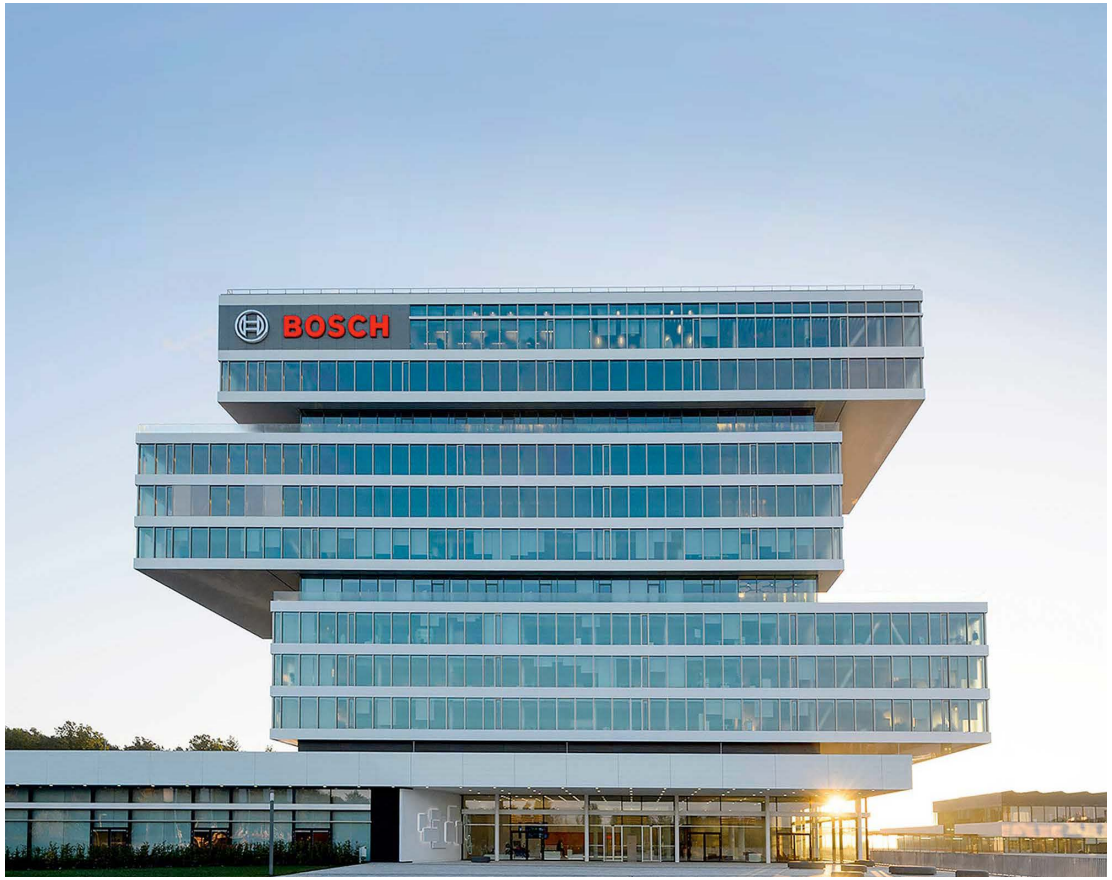
ELEVATE EMBEDDED REAL-TIME PROGRAMMING WITH A SYNCHRONOUS LANGUAGE

FRANZ-JOSEF GROSCH

JOINT WORK WITH FRIEDRICH GRETZ AND JENS BRANDT

BOSCH

Bosch – technology that is
“Invented for life”



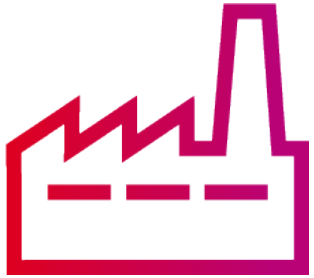
- Some 59,000[1] researchers and developers work at Bosch: at 120[2] locations worldwide, in a single network.
- Bosch is one of the world’s leading international providers of technology and services.
- Over the past six years, Bosch has invested more than 27 billion euros in research and development.
- Our objective: to develop innovative, useful, and exciting products and solutions to enhance quality of life – technology that is “Invented for life.”

[1] As of 12.16 [2] R&D locations with >50 associates, as of 12.16

Bosch – a global network
Four business sectors

Mobility Solutions

Industrial Technology



Energy and Building Technology

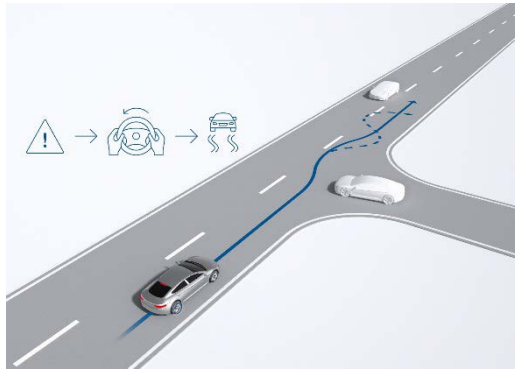
Consumer Goods

Franz-Josef Grosch | 1.10.2017

BOSCH

Bosch – technology to enhance quality of life
Examples

- ESP® – the Bosch anti-skidding system



- Engine Control – Gasoline direct injection

- Home appliances – Series 8 ovens

- Power tools – the Bosch Ixo

Franz-Josef Grosch | 1.10.2017

New Business Area
Bosch Sensortec – MEMS[1] Applications

Application areas of micro-electro-mechanical-systems, which enable the "feel" and "hear" of modern electronic devices

Mobile

Home Entertainment

Imaging

Sport & Fitness

Argumented Reality

Home Appliances

Industrial & Logistics

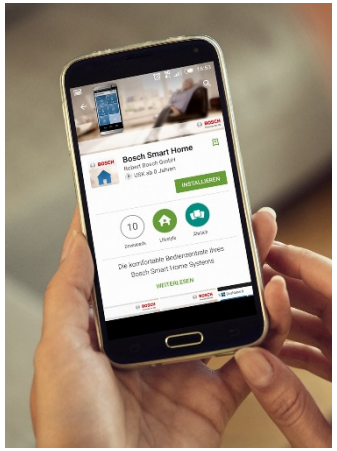
Smart Sensors

[1] micro-electro-mechanical-systems

Franz-Josef Grosch | 1.10.2017

BOSCH

Bosch Sensortec
XDK – A sensor platform for IoT Applications



Bosch IoT Suite

Bluetooth LE

Wi-Fi

Runtime environment

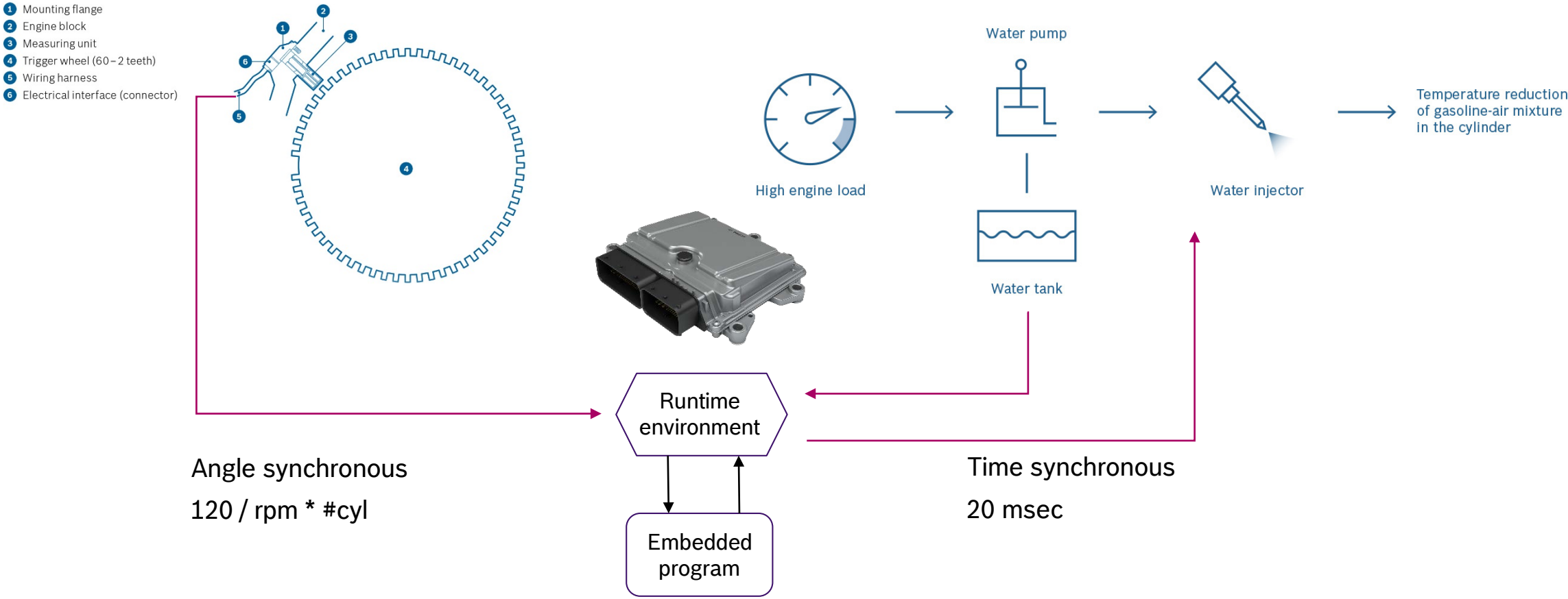
Embedded program

Franz-Josef Grosch | 1.10.2017

BOSCH

Mobility solutions

Control loops and mode switches – a peek into engine control

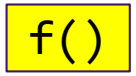


Franz-Josef Grosch | 1.10.2017

BOSCH

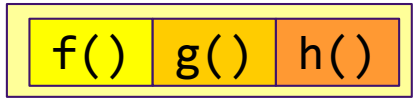
The structure of embedded software
Timing behaviour expressed via the environment

- "One-step" functions …



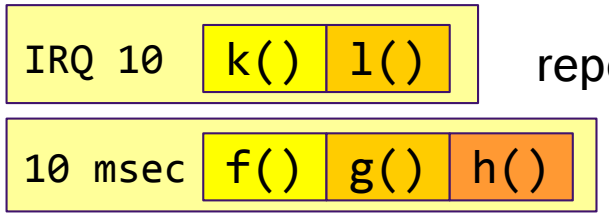
no inputs, no outputs, operates on global variables

- … composed in operating system tasks



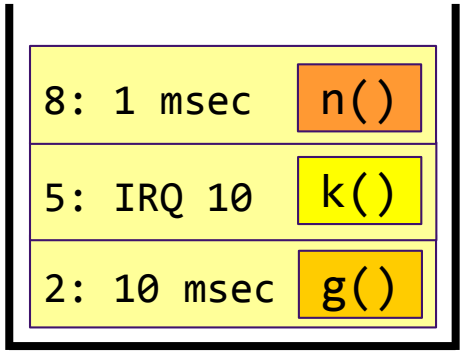
sequentially ordered

- … activated periodicly (time-triggered), sporadicly (event-triggered) or even rate-adaptive



repeated on clock-tick or on interrupt

- … scheduled according to priorities



high priority task pre-empts lower, task switch is a function call, only one stack for all tasks

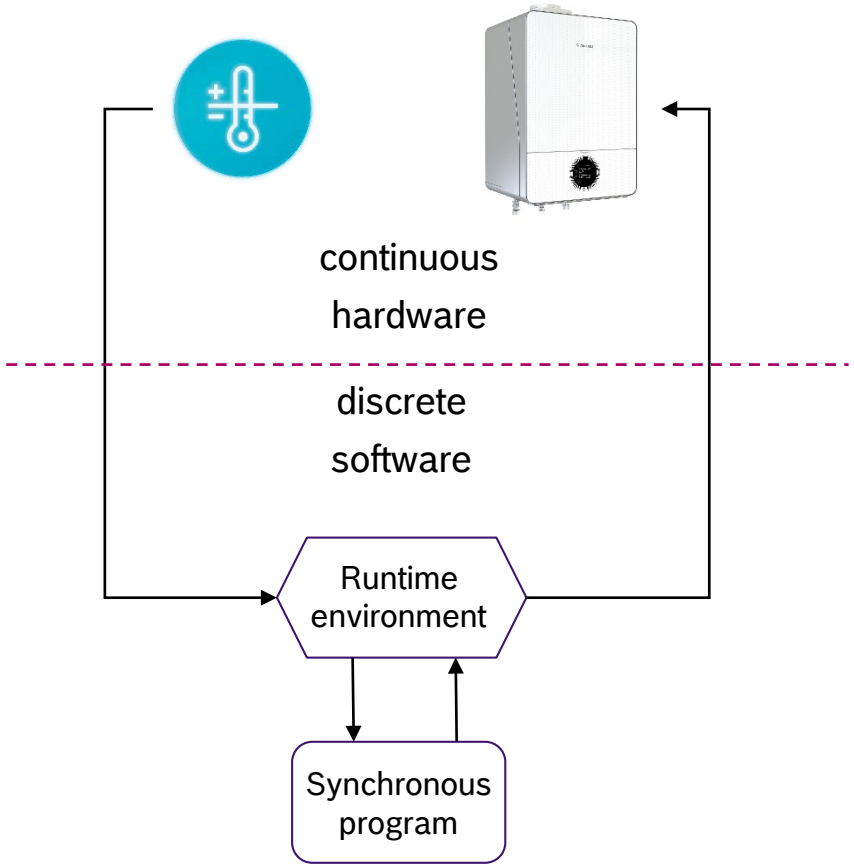
More details: *Real world automotive benchmark for free*, Kramer et al., 2015

The structure of embedded software
Questions causing trouble

- One-step functions
  - How do we manage state between two activations?
  - How do we reason about the behaviour of a function over repeated activations?
- Single task composition
  - Which function is writing what variable and when?
  - How do we reason about combinations of functions in a single task?
- Execution of parallel tasks
  - How is the dataflow between tasks?
  - How do we reason about combinations of functions in parallel tasks?

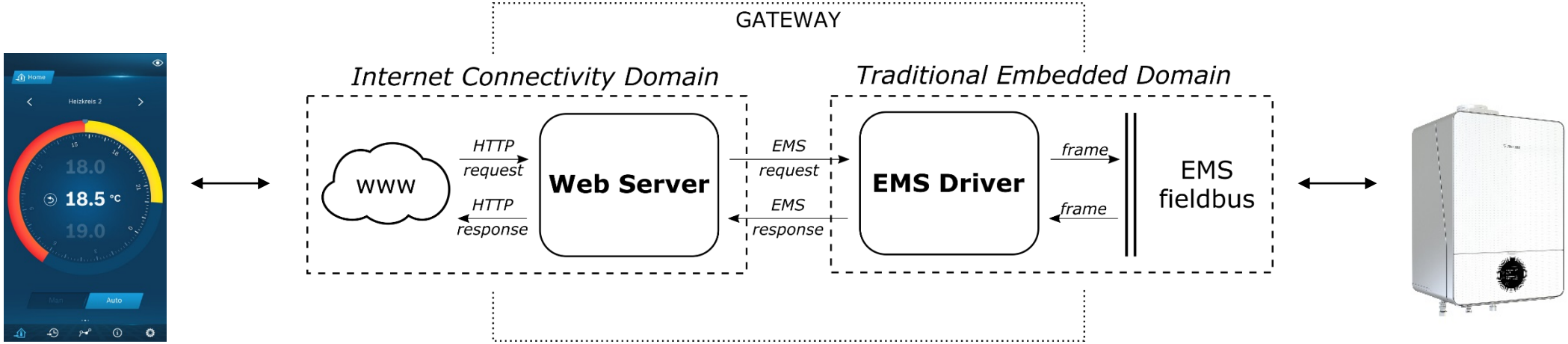
Franz-Josef Grosch | 1.10.2017

The synchronous paradigm
Can a synchronous language help?



- The environment translates the continuous physical world into discrete *reaction* triggers for the synchronous program
- The synchronous program is executed in *steps*
  - In every reaction the program executes one step
- Assume a step takes no time (happens instantaneously)
  - No change of input data throughout computation
- Code can be composed concurrently from *trails* (also called threads)
  - Accesses to shared data happen in a deterministic, causal order

Energy and Building Technology
An experiment with Céu



www.ceu-lang.org

Function-Oriented Decomposition for Reactive Embedded Software,
Matthias Terber, SEAA 2017

Elevate embedded real-time programming
Bridging the gap between models and C code

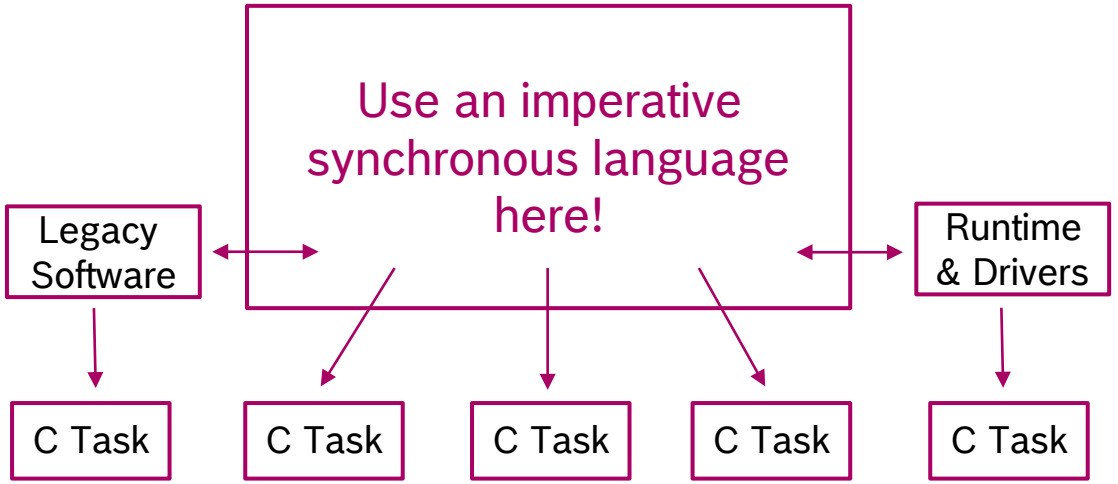
Analysis & Modelling

MODELICA Simulink® ASCET-CONGRA Scade® ASCET-DEVELOPER

Simulation & Transformation

Design & Implementation

Verification & Testing



Deployment

Hardware-in-the-loop

Bosch products



Field testing

Elevate embedded real-time programming
Our embedded software vision

- Support and attract software professionals
- Take care of multi-disciplinary engineering
- Enable clean embedded software design
- Re-enable reasoning about concurrent and parallel programs
- Improve productivity, agility, maintainability, testability, modularity, abstraction

Search for a synchronous language
Do we need to create our own language?

- Céu — purely event-triggered, no causality, soft-realtime
- Esterel — not really supported any more
- Lustre — not imperative, difficult to transfer as a textual language
- Quartz — focus too broad: specification of hardware and software

A suitable synchronous language
Language Requirements

- Software … not hardware
- Embedded … not “IT”-level software
- Real-time … time as a function, no blocking wait, no heap allocation
- Resource-constrained hardware … more static than dynamic
- Scalable … many lines of code
- Predictable … in terms of memory usage and execution time
- Multi-core … deterministic programming for parallel execution
- Legacy integration … call into and be called by legacy code

Express behaviour over time

Goal: Synchronous control for an imperative language

```
function times2 (x: int32) returns int32
    return x * 2
end

activity A (inA: int32)(outA: int32)
    repeat
        await true
        outA = times2(inA)
        if outA >= 0 then
            await inA > 0
        end
        outA = times2(inA)
    end
end
```

- Start with an imperative core language
  - Focus on readability
  - Safe saturation arithmetic, precisely sized types
- Add a statement to execute in steps
  - `await <condition/event/clock tick>`
  - `await true` ⇔ `await tick`
- Introduce two kinds of subprograms
  - `function` – one step, no await
  - `activity` – multiple steps, at least one await
- Introduce two kinds of parameter lists
  - Inputs – read only
  - Outputs – read/write

The Synchronous Paradigm
How is this executed?

```
function times2 (x: int32) returns int32
    return x * 2
end

activity A (inA: int32)(outA: int32)
    repeat
        await true
        outA = times2(inA)
        if outA >= 0 then
            await inA > 0
        end
        outA = times2(outA)
    end
end

var x: int32 = 0
var y: int32 = 0

run  A(x)(y)
```

	x->inA	outA->y
t_0	0	0
t_1	7	14
t_2	-1	14
t_3	5	28
t_4	-2	-8

time steps

A standard imperative core language implies

Sequentially Constructive Concurrency,

R. v. Hanxleden et al., 2013

The synchronous paradigm
How is this compiled

```
// C-like pseudocode
void mainloop () {
    step_of_A()
    ...
}
```

```
void step_of_A () {
    // restore code location
    // check await condition
    // execute corresponding computation
    // save location for next reaction
}
```

Boilerplate state management code

Hard to code manually

“Business” logic

Interesting part of the program

Use concurrent composition

Goal: Improve readability and flexibility

```
activity A(inA: int32)(outA: int32)
    ...
end

activity B(inB: int32)(outB: int32)
    ...
end

activity main()
    var x: int32 = 0
    var y: int32 = 0
    cobegin weak
        run A(x)(y)
    with
        run B(y)(x)
    end
end
```

- Add a control flow statement for concurrent composition
  - Focus on readability: `cobegin ... with ... with ... end`
  - Usable as an orthogonal statement
- Entering `cobegin` blocks (also called fork)
  - Execute multi-step trails (also called threads) concurrently
- Exiting `cobegin` blocks (also called join)
  - Terminate all trails in the same step
  - Strong trails run to their end, `weak` trails can be terminated early
- Execute in causal order of statement sequences
  - Concurrent `cobegin` blocks compile to sequential code
  - Causality analysis does not look into activities and functions
- Express parallel and/or
  - `cobegin ... with ... end` // parallel and
  - `cobegin weak ... with weak ... end` // parallel or

The synchronous paradigm
Causality Analysis

```
activity main ()
    var x: int32 = 0
    var y: int32 = 0
    cobegin weak
        run A(x)(y)
    with
        run B(y)(x)
    end
end
```

Error: causality cycle

Solve causality cycle →

```
activity main ()
    var x: int32 = 0
    var y: int32 = 0
    cobegin weak
        run A(x)(y)
    with
        run B(prev y)(x)
    end
end
```

Support separate compilation
Goal: Determine granularity of causality analysis

```
inline activity A(inA: int32)(outA: int32)
    ... outA = inA ...
end

inline activity B(inB: int32)(outB: int32)
    ... outB = 42 ...
end

activity main()
    var x: int32 = 0
    var y: int32 = 0
    cobegin weak
        ... y = x ...    // was: run A(x)(y)
    with
        ... x = 42 ...   // was: run B(y)(x)
    end
end
```

- `inline` activities make more programs causal

but

- `inline` prevents separate compilation

Use structured synchronous control flow
Preemptions complete control flow

```
enum Command
  Stop Start Accelerate Decelerate
end

activity controllingEngine(command: Command)
   when command == .Stop reset
   do
      when command == .Start abort
      run standingStill()

      repeat
         when command == .Accelerate abort
         run idling()

         when command == .Decelerate abort
         run running()
      end
   end
end
```

- Structured programming with synchronous control flow
  - Time steps: `await`
  - Concurrent composition: `cobegin`
  - Running multi-step behaviour: `run`
  - Preemptions: `abort`, `reset`, `suspend`

Programming for parallel execution
Extend synchronous programming for multi-core

- Logical execution time – deterministic communication
  - Always read the last known variable values (i.e. from the previous reaction)
  - Consistent view of data for all threads
- Not quite what we want
  - In general tasks can be unrelated
  - A complex analysis required to determine necessary amount of variable copies

From control models to real-time code using Giotto, Henzinger et al., 2003

- Clock refinement – rendezvous-based communication
  - A clock is refined by a child clock that has at least all the ticks of the parent (and more)
  - Synchronisation happens at common ticks
- Not quite what we want
  - Explicit knowledge of parent required
  - Child explicitly determines the point when the parent may react the next time

Clock refinement in imperative synchronous languages, Gemünde, Brandt, Schneider, 2013

Programming for parallel execution
Goal: Control flow and data flow across parallel tasks

```
@periodic(resolution = 1000)
clock _20_ms = count 20    // "parent" clock
clock _10_ms = _20_ms up 2 // upsampled refined clock

activity controlling() on _20_ms
    var input: float32
    var output: float32
    cobegin
        next run sensing()(next input) on _10_ms
    with
        repeat
            output = calculate(input) // control law
        await true end
    with
        next run actuating(prev output) on _10_ms
    end
end
```

- Introduce clocks
  - A clock represents a task (executed in parallel)
  - Code gets assigned to clocks (clock inference)
- Relate parent clocks and refined clocks
  - A refined clock contains all parent clock ticks
  - Synchronize at common ticks
- Cross-clock communication
  - Along clock refinement
  - According to logical execution time
- Cross-clock dataflow
  - Read from `prev`, write to `next` location
- Cross clock control flow
  - `run` and/or preempt activity in refined clock

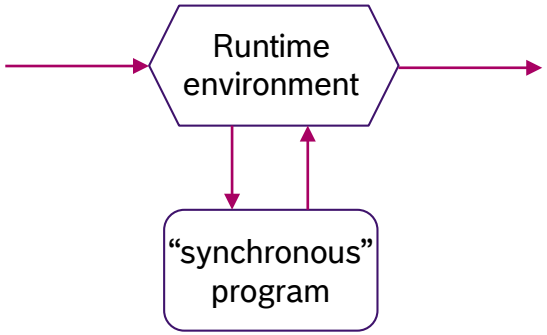
Structure and Design
Goal: Do not break causality analysis

```
struct Values
    first: int32 second: float32
end

type MyType()(dependOn: OtherType)
    param p = 1
    var v: int32 = 42

    function f() returns Values
        return {first = p * v, second = 0.0}
    end

    mutating activity a(inA: bool)
                       (outA: FurtherRefType)
        ...
    end
end

let otInstance = OtherType()
let mtInstance = MyType()(otInstance)
```

- Introduce two kinds of types
  - value types
  - reference types
- Introduce structured value types
  - Atomic for causality analysis
  - Useful for data exchange
  - `prev` and `next` allowed, shallow copying
- Introduce reference types
  - Atomic for causality analysis
  - Useful for structuring
  - Non-cyclic dependencies required
  - Bound during instantiation
- Introduce modules
  - Unit of separate compilation
  - Non-cyclic import hierarchy required

The runtime environment

Goal: Declarative description of the runtime environment




```
@periodic(resolution = 1000)
clock _20_ms = count 20

@entryPoint()
activity main()()
    ...
end
```

- Clocks
  - Mapping of base clocks
  - Management of refined clocks
- Interface Configuration
  - Sensors
  - Actuators
  - Networking interfaces
- Generators
  - Drivers
  - Events
  - Physical time

Globally asynchronous locally synchronous systems
Goal: Prevent over-synchronization

- Distributed programming and asynchronous coupling not in the focus
  - but necessary for large systems to prevent over-synchronisation
- Desynchronisation seems easier then synchronisation
- Optionally exclude physical values from causality analysis
  - Sensors are imprecise
  - physical values do not jump
- Communicate asynchronously via the environment
- Apply features like promises, futures, asynchronous calls

Blech – our imperative synchronous language
Tailored for embedded real-time programming

- **Behaviour over time**
- Programming of multi-step functions (`activity`)
- Execution in discrete time steps
- Zero-execution-time model within one time step

- **Concurrent composition**
- `cobegin` blocks
- Deterministic execution
- Causality analysis for a full-fledged language
- `prev` resolves causality issues

- **Synchronous control flow**
- Structured programming
- Preemptions: `abort, reset, suspend`
- `await`: condition, event, clock tick

- **Parallel execution**
- Truly parallel on multi-core HW
- Clocks as Tasks
- Cross-clock programming
- Based on LET and clock refinement
- `next` resolves buffering issues

- **Structure and design**
- Reference types
- Structured value types
- Events (not shown)
- Modules and separate compilation

- **Safety-related concepts**
- Reproducible timing behaviour
- No races, no deadlocks
- No recursion
- Static memory allocation, no pointers
- Generation of assertions

Where we stand
… and where to go

- We started to implement the compiler
- We are a small team
- We implement the compiler in F#

… we are open for discussion

… we are open for cooperation

… in the long-term we plan to go open-source

THANK YOU

www.bosch.com