SYNCHRONOUS LANGUAGES

LECTURE 14
THE BLECH PROGRAMMING LANGUAGE

16 JUNE, 2020

FRIEDRICH GRETZ
BOSCH CORPORATE RESEARCH

BOSCH

Today's speaker

Dr. Friedrich Gretz

Robert Bosch GmbH

Corporate Research in Renningen

Friedrich.Gretz@de.bosch.com

www.blech-lang.org

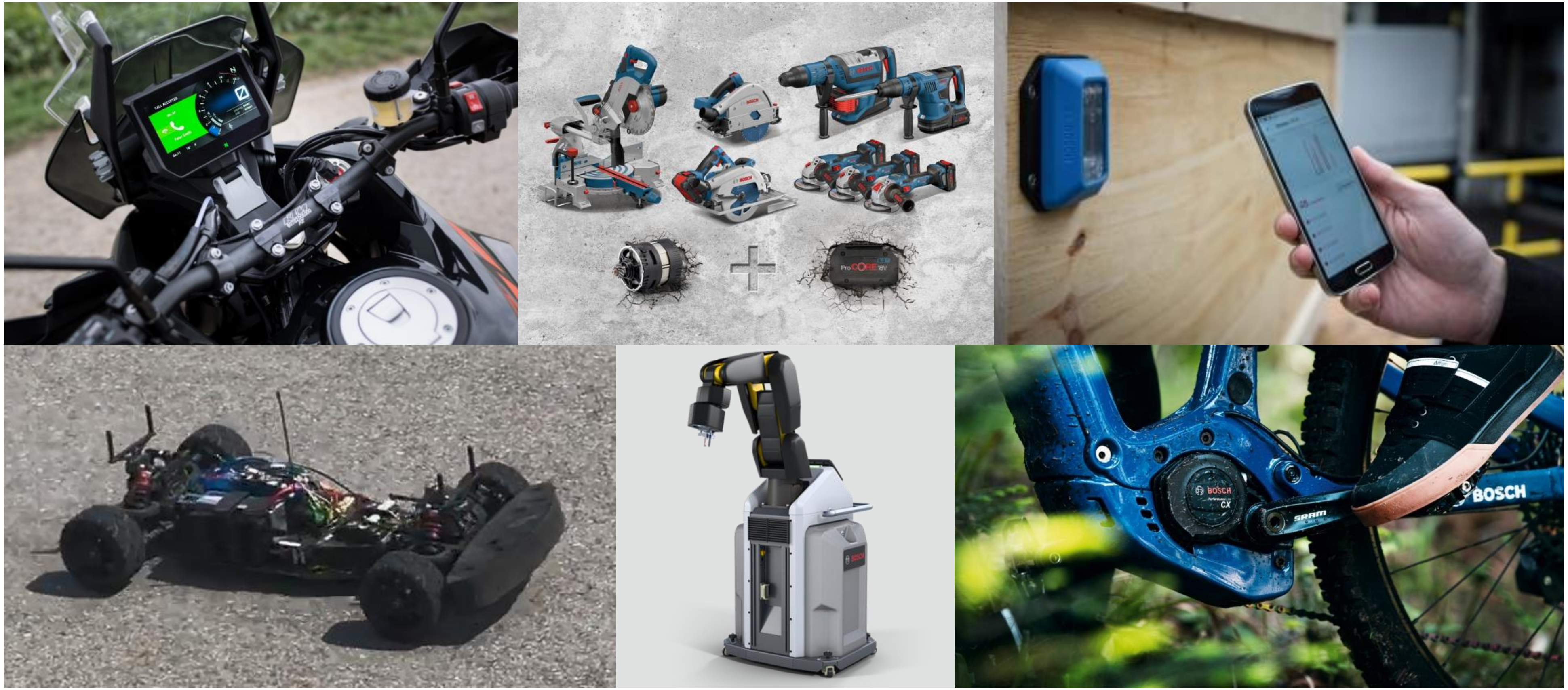
BOSCH

Overview

- Today's speaker
- **Why is synchronous programming interesting for Bosch?**
- Design goals
- Blech – as of now
- Application examples
- Outlook on planned features
- Additional remarks

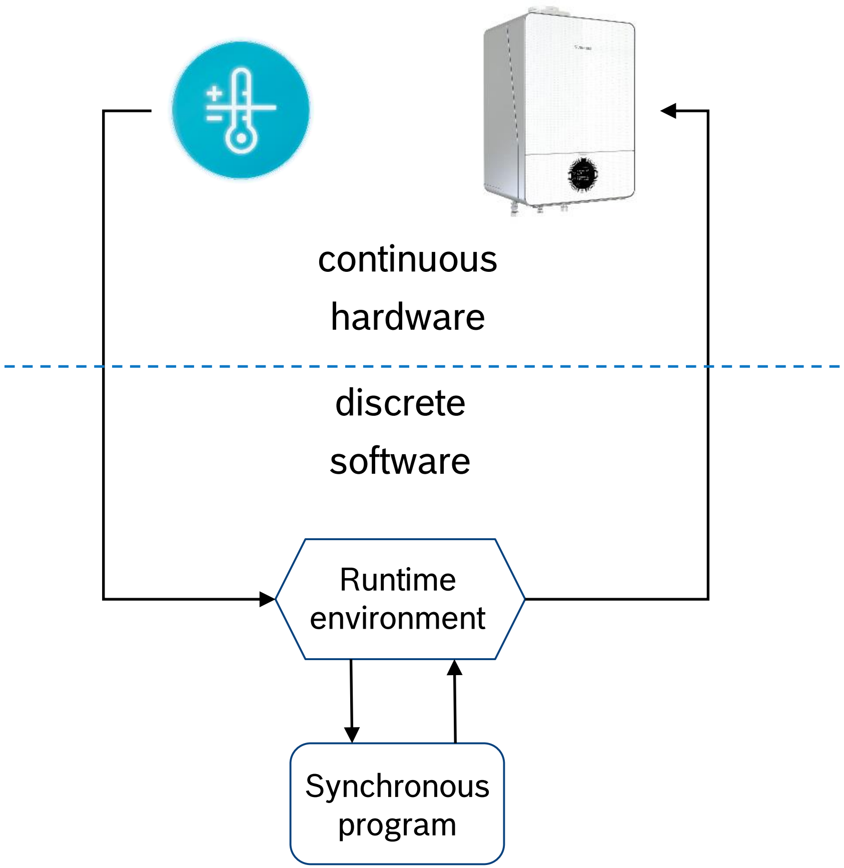
BOSCH

Why is synchronous programming interesting for Bosch?
Reactive, embedded software everywhere!



F. Gretz | 2020-06-15

Abstract view of a reactive system
Where do we use a synchronous language?



- Environment communicates asynchronously with physical world, drives synchronous programs
- A program is executed is *steps*
  - Assume a step takes no time (happens instantaneously)
  - No change of input data throughout computation
- A sequence of steps is called a thread of execution
- Threads can be composed concurrently
  - Accesses to shared data happen in a deterministic, causal order

Do we need a new synchronous language?
Available alternatives do not fulfill our requirements

▶ Céu	purely event-triggered, no causality, soft-realtime
▶ Esterel	no longer supported, not sequentially constructive, not separately compilable
▶ Lustre	not imperative, good for evaluating control loop equations but less intuitive for describing step-wise, mode switching behaviour
▶ SCCharts	automata centric view

Create a synchronous imperative language – Blech

Overview

- Today's speaker
- Why is synchronous programming interesting for Bosch?
- **Design goals**
- Blech – as of now
- Application examples
- Outlook on planned features
- Additional remarks

Design goals
Requirements

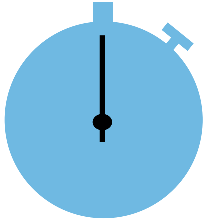
- Clear focus
  - Software
  - Reactive
  - Resource-constrained
  - Real-time
  - Scalable
- Domain orientation
  - Embedded
  - Control intensive systems
  - Computations and switching behaviour
  - Intertwined functionality
- Compatibility
  - Integration **of** legacy code
  - Integration **in** legacy code
  - Support separate compilation
- Deployment
  - Efficient code generation
  - Safe code generation
  - Integrate synchronous “execution shell” with existing real-time OS environments
  - Deployment on multi-core platforms
- Developer Orientation
  - Readable
  - Clear semantics
  - Stateflow in controlflow
  - Structured data
  - Code structuring, information hiding
  - Safe and modern type system
- Testing & Safety
  - Deterministic, repeatable testing
  - Integrate with existing simulation frameworks
  - Reduce false positives in static code analysis
  - Provide more guarantees, e.g. through causality

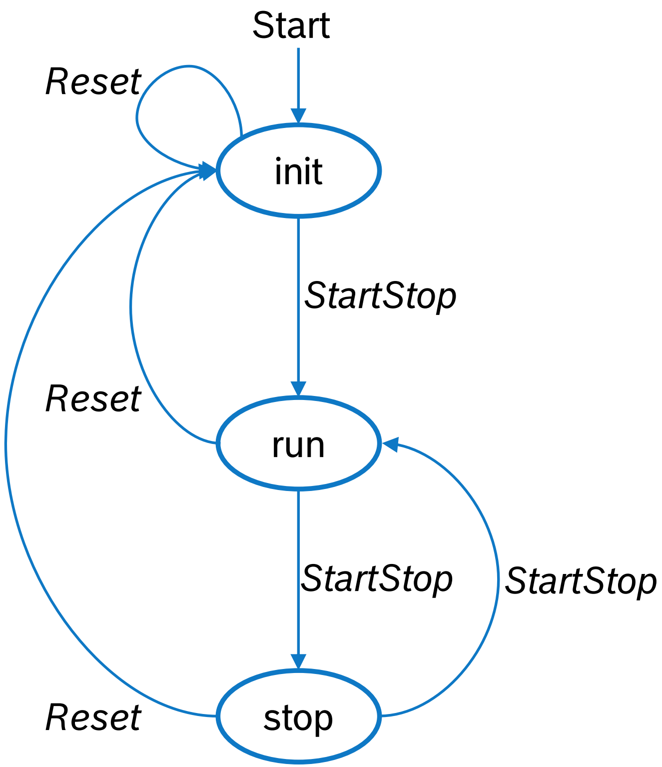
Overview

- Today's speaker
- Why is synchronous programming interesting for Bosch?
- Design goals
- **Blech – as of now**
- Application examples
- Outlook on planned features
- Additional remarks

Blech

Mode transitions as synchronous control flow






```
activity stopWatchControl (isPressedStartStop: bool,
                           isPressedReset: bool)
                          (display: Display)
   when isPressedReset reset
      // init
      resetToZero()(display)
      if not isPressedStartStop then
         await isPressedStartStop
      end
      repeat
         // run
         repeat
            await true
            increment()(display)
         until isPressedStartStop end
         // stop
         await isPressedStartStop
      end
   end
end
```


F. Gretz | 2020-06-15

Blech

Concurrent composition of behaviours over time

```
/// Main Program
@[EntryPoint]
activity Main (isPressedStartStop: bool,
               isPressedReset: bool)
   var display: Display
   cobegin // render
      repeat
         show(display)
         await true
      end
   with // control
      run StopWatchController(isPressedStartStop,
                              isPressedReset)
                             (display)
   end
end
```

- Execution model
  - Concurrent behaviours run in synchronised steps
- Causal order
  - first, update display data
  - second, show display
- Code generation
  - sequential code
  - Statically ordered by the compiler

Blech
Concurrency in detail

```
cobegin [weak]
    . . .
with [weak]
    . . .
end
```

do a step here

and

do a step there

```
cobegin
    run A(x)(z)
with
    run B(y)(x)
end
```

in every reaction:

"write before read!"

```
cobegin
    run A(x)(z)
with
    run B(y)(z)
end
```

write-write conflict

reject compilation

```
cobegin
    run A(x)(z)
with
    run B(z)(x)
end
```

write-read cycle

reject compilation

```
cobegin
    run A(x)(z)
with
    run B(prev z)(x)
end
```

solution

use previous value

F. Gretz | 2020-06-15

Blech

Concurrency in detail

```
…
cobegin
    …
with
    …
    cobegin
        run A(x)(z)
    with
        …
    end
    …
with
    run B(prev z)(x)
end
…
```

- Cobegin may have any fixed number of blocks
- Cobegin is orthogonal: it can be arbitrarily nested
- Subprograms are **black boxes** with interfaces, may be **compiled separately**
- **Interfaces** tell what data types are expected **and** whether data is only **read** or also **written**
- Causal **scheduling is dealt with locally** at call site
- Causality issues arise and may be debugged and **fixed within one cobegin statement**!

Blech

Concurrency in detail

cobegin statement terminates when…

```
cobegin
    run A(x)(z)
with
    run B(y)(x)
end
```

A **and** B have finished all their reactions

- start: cobegin, A, B
- A, B
- A, B; finished: A
- B; finished: B, cobegin

```
cobegin weak
    run A(x)(z)
with
    run B(y)(x)
end
```

B has finished all its reactions; A is possibly aborted

- start: cobegin, A, B
- A, B
- A, B; finished: A
- B; finished: B, cobegin

```
cobegin
    run A(x)(z)
with weak
    run B(y)(x)
end
```

A has finished all its reactions; B is possibly aborted

- start: cobegin, A, B
- A, B
- A, B; finished: A, B, cobegin

```
cobegin weak
    run A(x)(z)
with weak
    run B(y)(x)
end
```

A **or** B has finished all its reactions; the other one is possibly aborted

- start: cobegin, A, B
- A, B
- A, B; finished: A, B, cobegin

Blech
Use case for weak branches

runs indefinitely, unless terminated

```
cobegin weak
    run BlinkLED(...)(...)          // no arguments for readability
with
    run WaitForKeyStroke(...)(...) // no arguments for readability
end
```

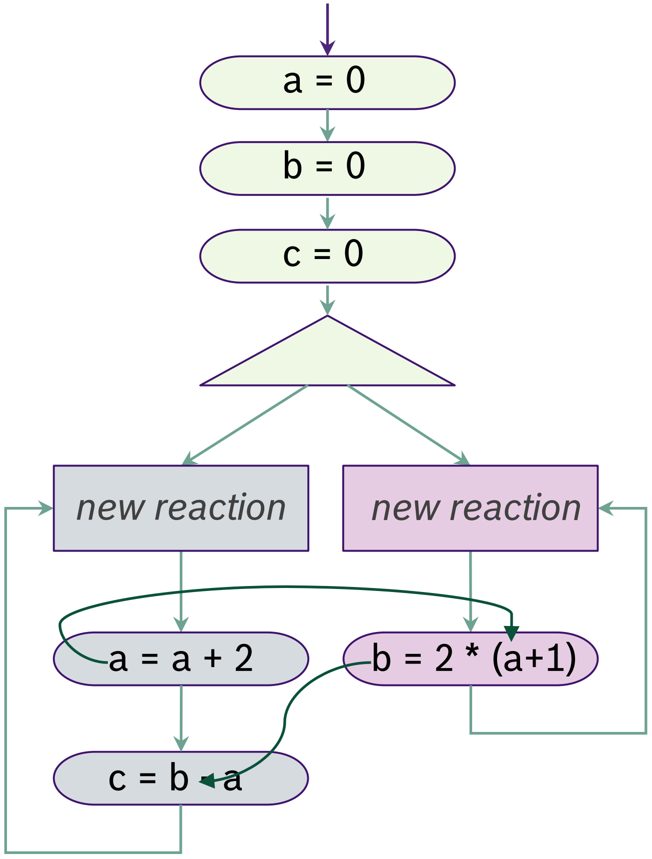
eventually terminates (if the system is to make any progress at all)

F. Gretz | 2020-06-15

Blech

Compiling activities to sequential C functions

```
@[EntryPoint]
activity main()()
  var a: int32
  var b: int32
  var c: int32
  cobegin
    repeat
      await true
      a = a + 2
      c = b - a
    end
  with
    repeat
      await true
      b = 2 * (a + 1)
    end
  end
end
```

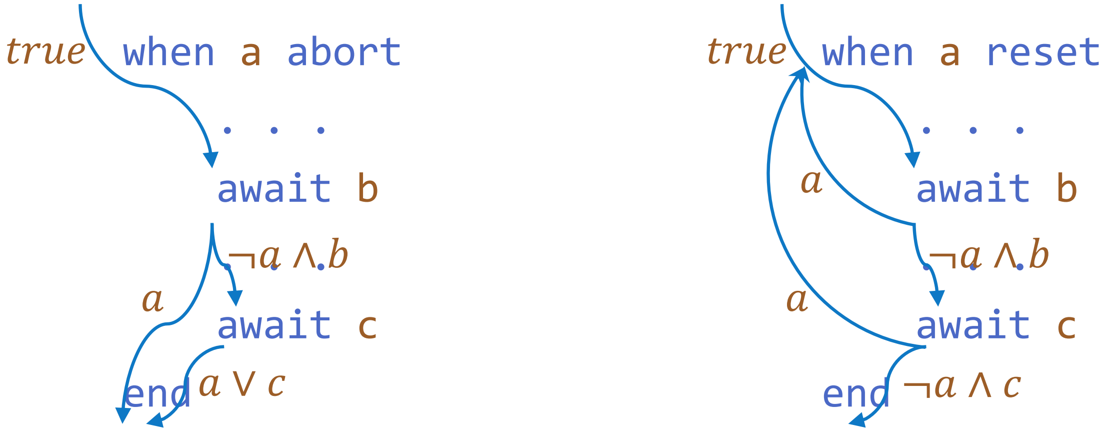



```
uint32_t main (int32_t *a, int32_t *b, int32_t *c, uint32_t *pc_1,
               uint32_t *pc_3, uint32_t *pc_2) {
  if ( *pc_1 == 2 ) {
    *a = 0;  // init
    *b = 0;
    *c = 0;
    *pc_2 = 7;  // enter branches and terminate step
    *pc_3 = 9;
    *pc_1 = 18;
  }
  if ( *pc_2 == 6 ) {  // left branch
    *a = (*a + 2);
    *pc_2 = 12;  // remember there is more to do
  }
  if ( *pc_3 == 8 ) {  // right branch
    *b = (2 * (*a + 1));
    *pc_3 = 9;  // terminate right step
  }
  if ( *pc_2 == 12 ) {  // left branch
    *c = (*b - *a);
    *pc_2 = 7;  // now terminate left step
  }
  _BLECH_SWITCH_TO_NEXTSTEP(*pc_2);  // bit-shifting magic
  _BLECH_SWITCH_TO_NEXTSTEP(*pc_3);
  _BLECH_SWITCH_TO_NEXTSTEP(*pc_1);
  return *pc_1;  // 0 means no more reaction steps to do
}
```

Blech

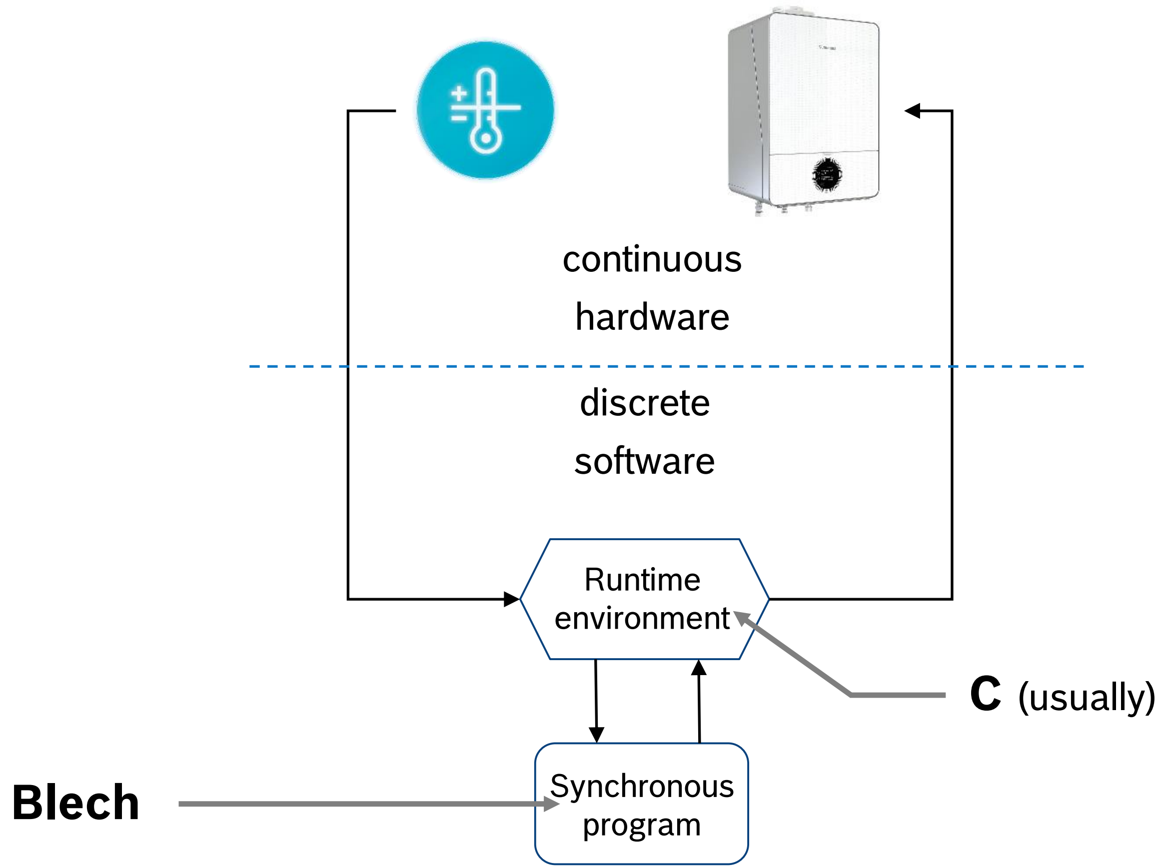
Stopping a behaviour

```
/// Keep blinking until the user presses button 1
activity Locked (pressedOne: bool) ()
    when pressedOne abort
        run Blink()
    end
end
```



F. Gretz | 2020-06-15

Abstract view of a reactive system
Where do we use a synchronous language?

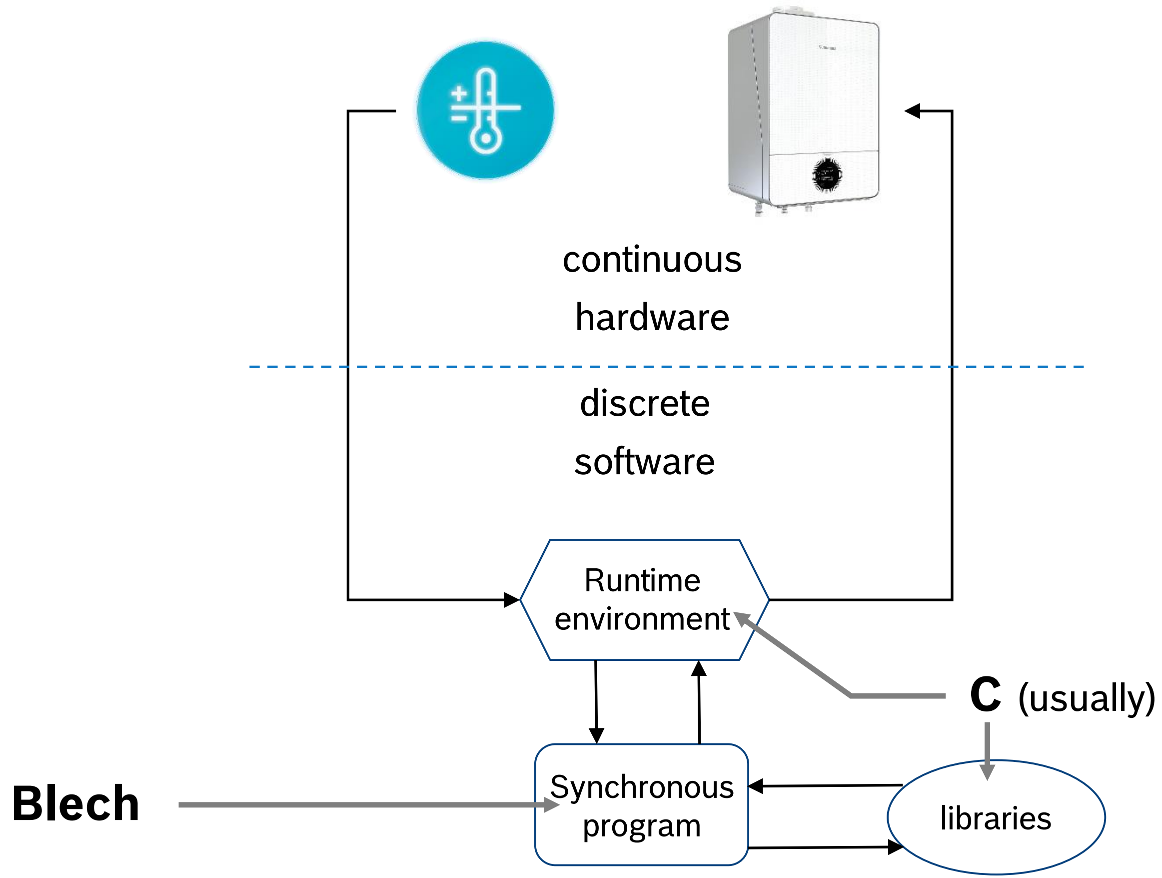


F. Gretz | 2020-06-15

C interoperability
Calling Blech from a runtime

```c
/* Main */

int main(int argc, const char * argv[])
{
    /* Create and initialize environment. */
    // …
    /* Initialize blech. */
    blc_blech_acc_init();

    /* Sense, control, act loop */
    while (1) {

        /* Get and adapt sensor input from environment. */
        env_input_state_t env_input_state = env_read(env);
        // …

        /* Run control reaction. */
        blc_blech_acc_tick(output_state.otherSpeed,
                           &output_state.egoSpeed,
                           &output_state.distance);

        /* Act on environment. */
        int hasCrashed = env_draw(env, &output_state);
        // …

        /* Wait for next tick.*/
        usleep(update_frequency);
    }

    /* Destroy environment. */
    env_destroy(env);

    return 0;
}
```

Abstract view of a reactive system
Where do we use a synchronous language?



F. Gretz | 2020-06-15

C interoperability
External constants

C

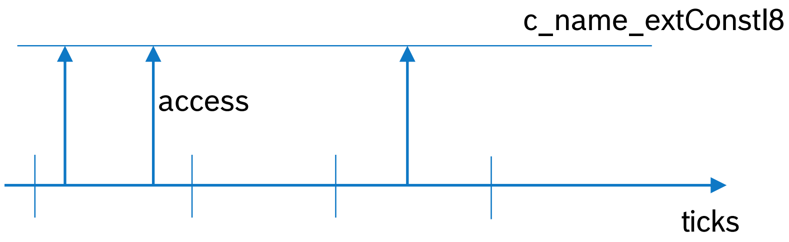
```
#define c_name_extConstI8 8
```

Blech

```
@[CConst(binding="c_name_extConstI8",
         header="my_externals.h")]
extern const extConstI8: int32
```

assumption

is constant throughout the whole runtime



usage

```
function f ()
    let testI8 = extConstI8
    // ...
end
```


F. Gretz | 2020-06-15

C interoperability
External volatile read-only memory

C

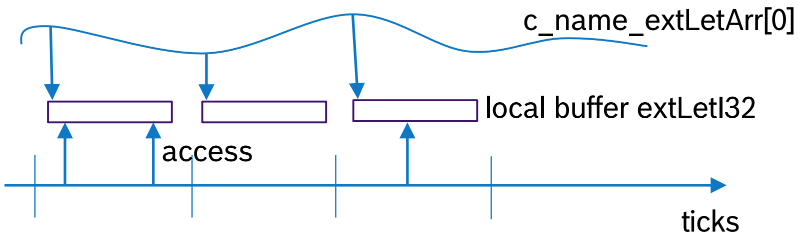
```
int c_name_extLetArr[8];
```

Blech

```
activity B ()
    @[CInput(binding="c_name_extLetArr[0]",
             header="my_externals.h")]
    extern let extLetI32: int32
    ...
end
```

assumption

is volatile



usage (multiple concurrent instances of B may run)

```
cobegin
    run B  ()
with
    run B  ()
end
```

C interoperability
External volatile read-write memory

C

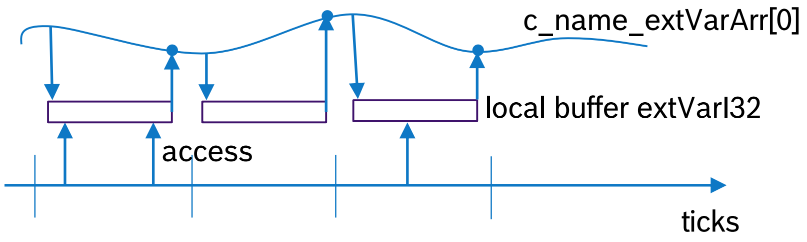
```
int c_name_extVarArr[8];
```

Blech

```
activity B ()
    @[COutput(binding="c_name_extVarArr[0]",
              header="my_externals.h")]
    extern var extVarI32: int32
    ...
end
```

assumption

is volatile



usage (B is a singleton now)

```
cobegin
    run B  ()
with
    run B  () ←—— error!
end
```


F. Gretz | 2020-06-15

C interoperability
External (singleton) functions

C

```
uint8_t NRF24L01_spiIsReady (void)
{
    return (HAL_SPI_GetState(nrf24l01_init.hspi)
            == HAL_SPI_STATE_READY) ? 1 : 0;
}
```

Blech

```
@[CFunction(binding = "NRF24L01_spiIsReady",
            header = "nrf24l01.h")]
extern singleton function spiIsReady () returns bool
```

assumption

singleton:

- function either reads a volatile value
- or has a side-effect on the environment

not singleton:

- re-entrant, side-effect free function

usage (spiIsReady is declared to be a singleton)

```
cobegin
    await spiIsReady()
with
    await spiIsReady()   ←—— error!
end
```


F. Gretz | 2020-06-15

Blech

Find all details on the language as it is currently implemented at

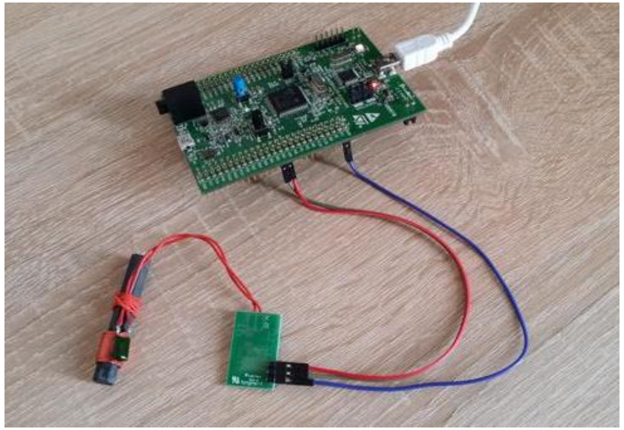
https://www.blech-lang.org/docs/user-manual/

If you find any mistakes or lack of clarity, please do notify us via Github issues.

Overview

- Today’s speaker
- Why is synchronous programming interesting for Bosch?
- Design goals
- Blech – as of now
- **Application examples**
- Outlook on planned features
- Additional remarks

Application examples

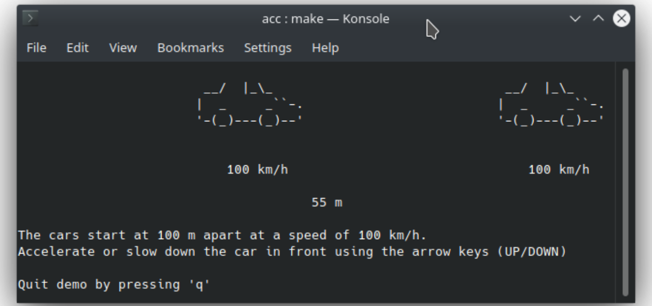


DCF 77 signal decoding

bare metal

"Virtual lock"

FreeRTOS + Mita



Homework: ACC

Linux OS + ncurses

Controller development

MATLAB/Simulink

(S-function)

BOSCH

Overview

- Today’s speaker
- Why is synchronous programming interesting for Bosch?
- Design goals
- Blech – as of now
- Application examples
- **Outlook on planned features**
- Additional remarks

Outlook on planned features
What else should be possible with Blech?

Mechanisms

- Parallel programming with multiple clocks
- Event communication using signals

Software Engineering

- Module system
- Immutable references

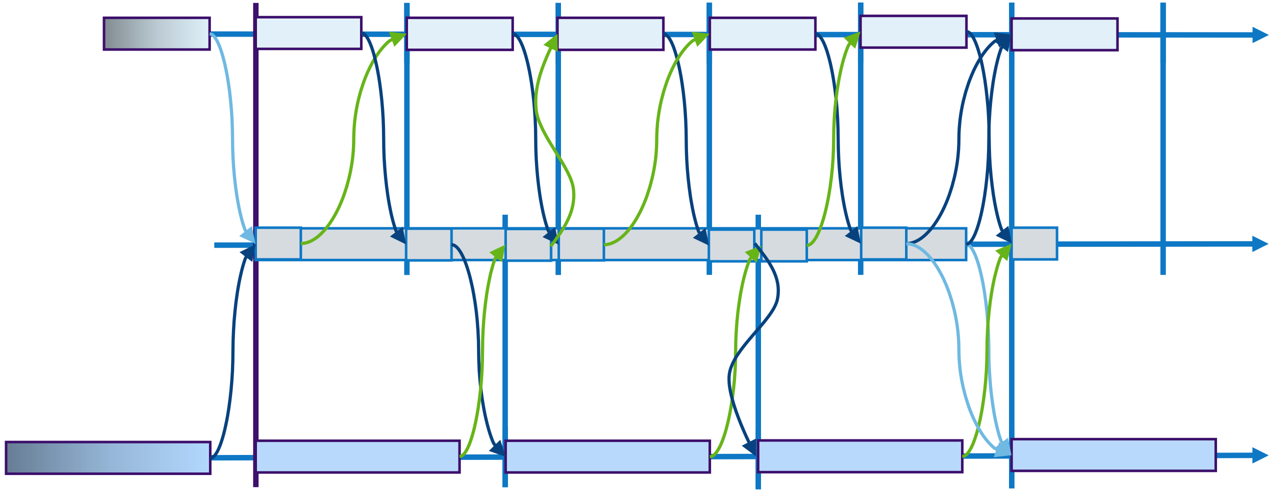
Safety

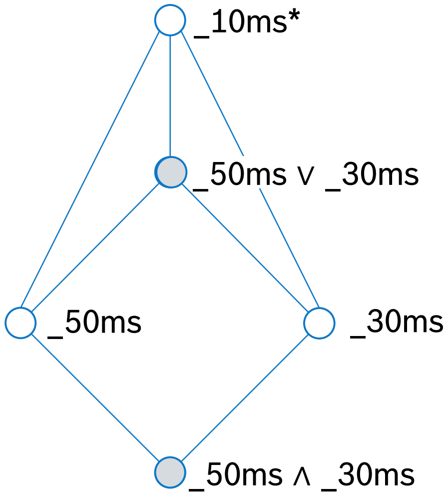
- Physical dimensions
- Safe code generation

F. Gretz | 2020-06-15

Mechanisms
Parallel programming with multiple clocks

- Communicating tasks must have related clocks
- Communication is done by sampling according to logical execution time
- Deterministic, consistent, compositional, real-time capable





F. Gretz | 2020-06-15

Mechanisms

Communicating events with signals

```
activity Signalling()
    var finished: signal

    cobegin
        run anActivity()
        emit finished
    with
        repeat
            ...
            await true
        until finished end
        ...
    end
end
```

Signal

- Presence flag
- Optional payload
- Only present in emitting time step
- Automatically absent after reaction

F. Gretz | 2020-06-15

BOSCH

Software engineering
Module system

- Decompose code into separately compiled units: “modules” (do not confuse with Esterel modules!)
- Modules must export types, activities or functions that should be used by their clients (API, information hiding)
- Interfaces must take causality information into account
- Module system translates names to unique C identifiers (everything is globally visible in C)

Safety
Physical dimensions

```
unit m
unit s

var length: float32[m]
var duration: float32[s]

length = 2 * length // ok
length = 2 + length // error!

let speed = length / duration // ok
let nonsense = length + duration // error
```

- The physical dimension are part of the data type
- Machine data types prevent arithmetic operations on incompatible types
- Physical dimensions prevent arithmetic operations which do not make sense (cf. homework code)

Safety

Safe code generation

```
let a: [7]float32 = {...}
...
let x = a[i] // ok, provided i >= 0, i <= 6
```

Debug code generation

```
float x;
if(i >= 0 && i <= 6) {
  x = a[i];
} else {
  haltWithDebugInfo();
}
```

Release code generation

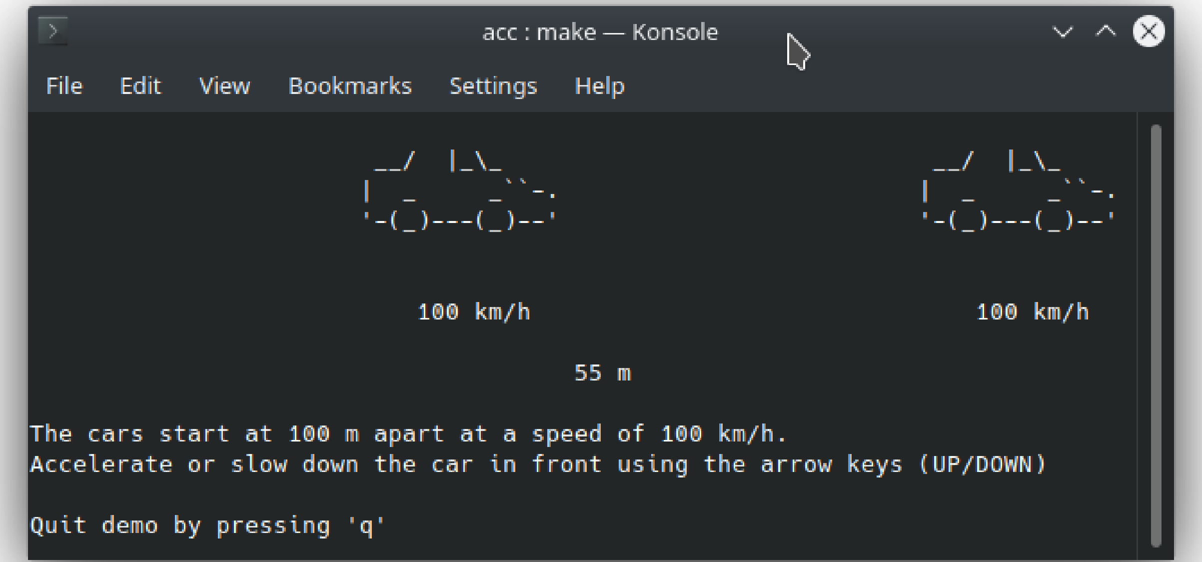
```
float x;
if(i >= 0) {
  if (i <= 6) {
    x = a[i];
  } else {
    x = a[6];
  }
} else {
  x = a[0];
}
```


F. Gretz | 2020-06-15

Overview

- Today’s speaker
- Why is synchronous programming interesting for Bosch?
- Design goals
- Blech – as of now
- Application examples
- Outlook on planned features
- **Additional remarks**

Homework
Adaptive Cruise Control



F. Gretz | 2020-06-15

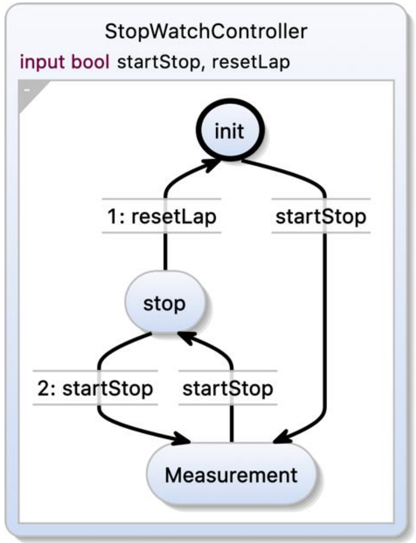
Bachelor / Master Thesis

Extraction of mode diagrams from Blech

Get in touch with Prof. von Hanxleden

```
activity StopWatchController (startStop: bool, resetLap: bool)
                             (display: Display)
    var totalTime: int32
    var lastLap: int32
    repeat
        totalTime = 0     // State init
        lastLap = 0
        writeTicksToDisplay(totalTime)(display)
        await startStop   // Transition init -> run
        repeat
            cobegin weak
                await startStop
            with weak
                run Measurement(resetLap)
                               (totalTime, lastLap, display)
            end
            // State stop, show total time and wait
            writeTicksToDisplay(totalTime)(display)
            await startStop or resetLap
            // Run again if only startStop was pressed
        until resetLap end // Back to init if
    end                    // resetLap was pressed
end
```

⇒



F. Gretz | 2020-06-15

Where do I get Blech?
or how do I participate?

All info is available at www.blech-lang.org

Blech is **open source**! Driven by Bosch CR.

Try Blech right now, start with tutorials and other examples. Why not write a blog post about your experience?

Participate in discussions and give feedback on language design

Actively shape Blech by contributing to the compiler, tooling or documentation

Let's collaborate on product software, an evaluation prototype, a student thesis or internship

We happily give a talk for your developers or managers or organise a hands-on tutorial

The Blech team is open for ideas and discussions

BOSCH

Where do I get Blech?
or how do I participate?

Get in touch with us:

Friedrich.Gretz@de.bosch.com
Franz-Josef.Grosch@de.bosch.com