BLECH –
A SYNCHRONOUS LANGUAGE
FOR EMBEDDED REAL-TIME PROGRAMMING

KEYNOTE, WCET 2019

FRANZ-JOSEF GROSCH

JOINT WORK WITH FRIEDRICH GRETZ
Bosch
Technology to enhance quality of life

Bosch is one of the world’s leading international providers of technology and services.

Engineering locations worldwide, in a single network.

Over the past years, Bosch has invested several 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”.
Bosch – a global network
Research and Development 2018

4 business sectors
- Mobility Solutions
- Industrial Technology
- Consumer Goods
- Energy & Building Technology

In 2018
78.5 billion €
Company’s sales

In 2018, Bosch invested
7.3 billion €
in research and development

The international research network of Corporate Research has
12 locations in 8 countries

130
Bosch research and development locations worldwide

Corporative Research & BCAI

392.4 million €
were invested in Corporate Research & Bosch Center for Artificial Intelligence

69,000
Bosch researchers and developers worldwide

~1,800
associates in Corporate Research & Bosch Center for Artificial Intelligence

Franz-Josef Grosch | 2019-07-09
WCET 2019
© Robert Bosch GmbH 2018. All rights reserved, also regarding any disposal, exploitation, reproduction, editing, distribution, as well as in the event of applications for industrial property rights.
Technology to enhance quality of life

Some examples

- Driver assistance and automated driving
- Powertrain systems and electrified mobility
- Home appliances
- Power tools
Bosch “Things” in a connected world

The importance of embedded software

- Bosch's biggest strength in the IoT ecosystem are the Bosch “Things”

- These devices and physical products cover a multitude of domains

- Each with high market penetration typically among the TOP 3

- “Bosch is a giant in embedded software”
  (Dr. Volkmar Denner, CEO)
The structure of embedded software

Timing behaviour expressed via the environment

- “One-step” functions ...

- ... composed in operating system tasks

- ... activated periodic (time-triggered), sporadic (event-triggered) or even rate-adaptive

- ... scheduled according to priorities

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?
  - What is a suitable order of functions in a task?
  - How do we reason about combinations of functions in a task?

- Execution of parallel tasks
  - How is the dataflow between tasks?
  - How do we reason about combinations of functions in parallel tasks?

Do we need a programming language better suited to embedded requirements?
Why a new language?
Build a better tool!
Should the language be synchronous?

The synchronous paradigm

- Environment communicates asynchronously with physical world, drives synchronous programs

- A program is executed is steps
  - A sequence of steps is called a thread (we prefer trail)

- Assume a step takes no time (happens instantaneously)
  - No change of input data throughout computation

- Sequences of steps 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, focus on control flow and coordination
- Lustre not imperative, focus on data flow, difficult to transfer
- Quartz focus too broad: specification of hardware and software

Create a synchronous imperative language - Blech
Goal: Synchronous control for an imperative language
Express behaviour over time

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)
    until outA < 0 end
end

- Start with a safe imperative core language
  - Focus on readability
  - Safe saturation arithmetic, precisely sized types
  - No global variables

- 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
How is this executed?
Stackless execution in macro steps

<table>
<thead>
<tr>
<th>Time steps</th>
<th>inA</th>
<th>outA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{i+1}$</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>$t_{i+2}$</td>
<td>-1</td>
<td>14</td>
</tr>
<tr>
<td>$t_{i+3}$</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>$t_{i+4}$</td>
<td>-2</td>
<td>-8</td>
</tr>
</tbody>
</table>

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)
   until outA < 0 end

A standard imperative core language implies
Sequentially Constructive Concurrency, Hanxleden et al., 2013
How is this compiled?
Functions called on every step

// 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
How is this developed?

Mode transitions as synchronous control flow

activity stopWatchControl (isPressedStartStop: bool,
isPressedReset: bool)
(display: Display)

reset when isPressedReset before
// init
display:resetToZero()
if not isPressedStartStop then
  await isPressedStartStop
end
repeat
  // run
  repeat
    await true
    display:increment()
  until isPressedStartStop end
// stop
  await isPressedStartStop
end
end
How is this composed?
Concurrent composition of behaviours over time

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

- Execution model
  - Concurrent behaviours run in synchronized steps

- Causal order
  - first, update display data
  - second, show display

- Code generation
  - sequential code
  - Statically ordered by the compiler
Combine behaviours over time
Concurrent composition with improved readability and flexibility

- Add a control flow statement for concurrent composition
  - Focus on readability: \texttt{cobegin ... with ... with ... end}
  - Usable as an orthogonal statement

- Entering \texttt{cobegin} blocks (also called fork)
  - Execute multi-step trails (also called threads) concurrently

- Exiting \texttt{cobegin} blocks (also called join)
  - Terminate all trails in the same step
  - Strong trails run to their end, \texttt{weak} trails can be terminated early

- Execute in causal order of statement sequences
  - Concurrent \texttt{cobegin} blocks compile to sequential code
  - Causality analysis does not look into activities and functions

- Express parallel and/or
  - \texttt{cobegin ... with ... end} \hspace{1em} // parallel and
  - \texttt{cobegin weak ... with weak ... end} \hspace{1em} // parallel or

\begin{verbatim}
activity A(inA: int32)(outA: int32) ...
end

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

activity main()
  var x: int32
  var y: int32
  cobegin weak
    run A(x)(y)
    with
      run B(y)(x)
  end
end
\end{verbatim}
Deterministic sequential execution of concurrent code
Non-global causality analysis

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

deterministic sequential execution of concurrent code

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

Error: causality cycle
Solve causality cycle
How is this compiled?

Local variables stored in global memory

Activity A(inA: int32)(outA: int32)
  cobegin
  run C()
  with
  run D()
  end
end

Activity B(inB: int32)(outB: int32)
  run C()
  run D()
end

Activity main ()
  var x: int32
  var y: int32
  cobegin weak
    run A(x)(y)
    with
    run B(prev y)(x)
  end
  run E()
end

Pre-computed "cactus stack"

```c
struct A{
  /* A's locals */
  struct C c;
  struct D d;
};
struct B{
  /* B's locals */
  union {
    struct C c;
    struct D d;
  }
};
struct Main{
  /* Main's locals */
  union {
    struct {
      struct A a;
      struct B b;
    } a_with_b;
    struct E e;
  }
};
struct Main _Globals;
```

Call graph

"Stack views"
Software structure and design
Structured data types, references, objects, modules

```
struct Value
    var first: int32
    var second: float32
end

ref struct MyType
    var flag: bool
    var ref value: Value // initialised at declaration
end

with
    const c: int32 = 42 // compile time constant
    param p: float32 = 9.81 // hex file constant

function f() returns int32 // static subprogram
end

mutating activity mt: actMethod() // method subprogram
    mt.value.first = f() // deref 'value' taken automatically
end

var v: Value = {first = 1} // second gets default value
var mt: MyType =
    {flag = true, value = v} // ref 'v' taken automatically
```

- 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
Clocks – a way to express multi-form time
Speed – a synchronous “Hello world”

```
activity countingCmBetweenSeconds()(distance: int32) on both
    repeat await true // any tick
        if tick cm then
            distance = distance + 1
        elseif tick sec then
            distance = 0
        end
    end

activity updatingSpeed(distance: int32)(speed: int32) on both
    repeat await tick sec
        speed = distance
    end

activity startup() on both
    var distance: int32 = 0
    var speed: int32 = 0
    cobegin
        run countingCmBetweenSeconds()(distance)
        with
            run updatingSpeed(distance)(speed)
    end
```
Parallel programming with clocks
Logical execution time and clock refinement

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

Clock refinement in imperative synchronous languages, Gemünden, Brandt, Schneider, 2013
Simplified static analysis
The compiler knows and guarantees static properties

- No recursion
- No pointers
- No address arithmetic
- No dynamic allocation
- No concurrent write conflicts
- No dynamic concurrency
- No dynamic parallelism
- No global variables
- No undefined values
- No programmer-defined locking
- Separate compilation

- Predictable memory usage
- Predictable execution time
- Always one writer multiple readers
- Statically known end-to-end latencies
- Statically known number of clocks
- Known, possible number of tasks
- Predictable synchronisation effort
- Easier task deployment
- Easier variable mapping
- Room for optimisation in code generation
- Reduced need for whole program analysis
“Bosch is a giant in embedded software” (Dr. V. Denner, CEO)

Wishlist for an embedded real-time programming language

- Hybrid: Time-driven and event-driven
- Predictable and deterministic
- Synchronous concurrency
- Hard real-time
- Bounded memory usage and execution time
- Easy integration of C code
- Prepared for multi-core
- Explicit control of deployment and variable placement
- Compile-time mechanisms for structuring and variants
- Safe shared memory
- Safe type system
- Expressive and productive
- A “real cool” development environment

Core Business
“Things” driven by embedded software
Elevate embedded real-time programming
Bridging the gap between models and C code

Analysis & Modelling
- ASCET-DEVELOPER
- SCODE-CONGRA
- Simulink®
- MODELICA
- Scade®

Design & Implementation
- Real-time requirements
- Reactive concerns
- Software design
- Built-in concurrency
- Deterministic parallelism

Deployment
- C Task
- Legacy Software
- Runtime & Drivers

Verification & Testing
- Assertion checking
- Unit testing
- Debugging
- Closed-loop simulation

Hardware-in-the-loop
- Bosch products

Field testing
Elevate embedded real-time programming

Our embedded software vision

- Take care of multi-disciplinary engineering

- Express timing behavior in the program (not in the environment)

- Enable clean embedded software architectures

- Re-enable reasoning about parallel programs

- Improve productivity, agility, maintainability, testability, modularity, abstraction

- Support and attract software professionals
First steps on a “cool” development environment
A Blech Language Server used with Visual Studio Code
Where we stand
... and where to go

- We have a clear vision of Blech’s features

- We are a small team

- We implement the compiler, the language server and the build system in F#

... we are open for discussion

... we are open for cooperation

... we prepare to go open-source
THANK YOU